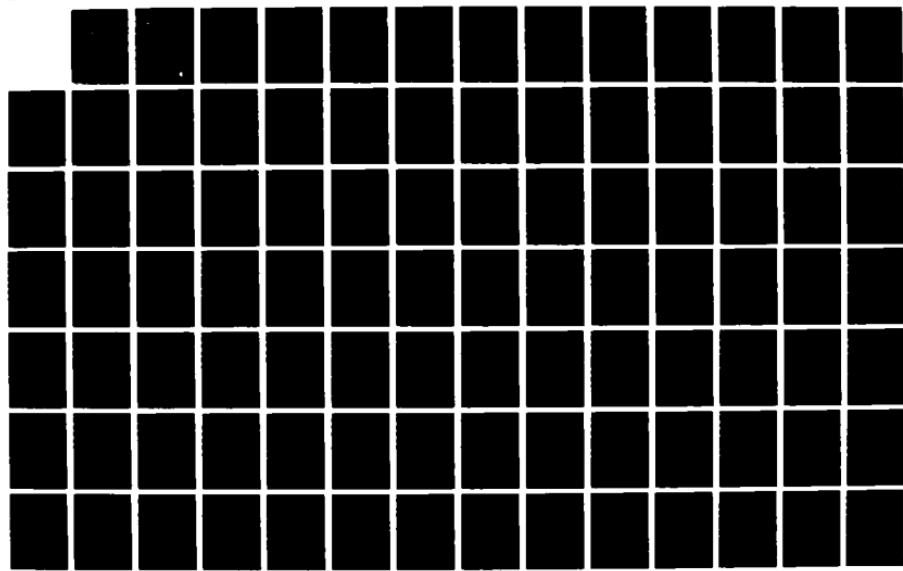


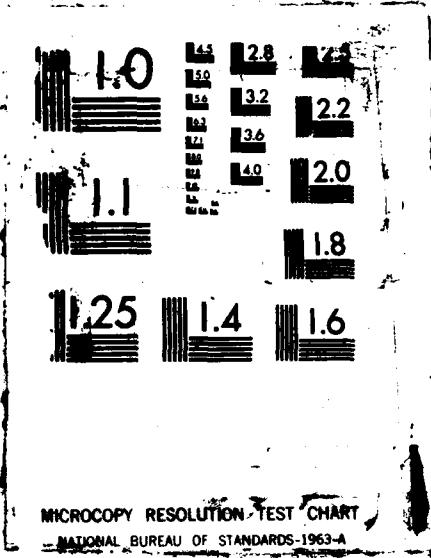
AD-A186 693 SECURE DISTRIBUTED PROCESSING SYSTEMS(U) CALIFORNIA 1/4
UNIV LOS ANGELES SCHOOL OF ENGINEERING AND APPLIED
SCIENCE G J POPEK DEC 78 UCLA-ENG-7955

UNCLASSIFIED MDA903-77-C-0211

F/G 12/7

NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

AD-A186 683

UCLA-ENG-7955
SDPS-78-004
DECEMBER 1978



SECURE DISTRIBUTED PROCESSING SYSTEMS: QUARTERLY
TECHNICAL REPORT, JULY 1978 - DECEMBER 1978

DTIC FILE COPY

GERALD J. POPEK, PRINCIPAL INVESTIGATOR

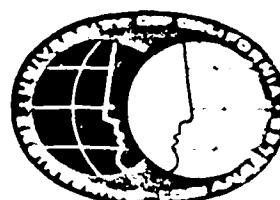
SECURE SYSTEMS AND SOFTWARE
ARCHITECTURE GROUP

DTIC
SELECTED
OCT 21 1987
S **D**
CoD

Approved for public release; distribution unlimited

COMPUTER SCIENCE DEPARTMENT

School of Engineering and Applied Science
University of California
Los Angeles



87 108023

SECURE DISTRIBUTED PROCESSING SYSTEMS

**Advanced Research Projects Agency
Quarterly Technical Report**

July 1978 - December 1978

Introduction

This technical report covers research carried out by the Secure Distributed Processing Systems group at UCLA under ARPA Contract MDA-903-77-C-0211 during the last two quarters of 1978. Progress has been made on all four contracted tasks, namely network security, data management security, high availability secure information management, and UCLA secure system enhancements. Below, we describe that progress.

Task I - Network Security

As part of UCLA's participation in the larger ARPA sponsored network security experiment employing BCR units to demonstrate that end to end encryption of individual connections on the ARPANET is feasible, a BCR unit has been successfully installed and checked out at UCLA with the help of Collins Radio personnel.

Additional work was also done to understand the limitations of public key based encryption algorithms, especially in their application to digital signatures. After the inherent limitations were understood, a variety of methods were developed, using either public key or conventional algorithms, which are more robust than the unadorned public key approach.

Task II - Data Management Security

The kernel implementation for the Ingres database system, which was begun in previous quarters, was successfully completed. The resulting prototype kernel provides secure operation for the normal functions of retrieval and update. Performance of the kernel based system is excellent, comparable to that of the unaltered Ingres system itself. This results makes it quite clear that kernel based secure data management systems are feasible. Additional work is needed, however, to demonstrate how various maintenance functions can be supported under the security constraints.

Task III - High Availability Secure Information Management

Significant results were completed during this contract period. Protocols for reliable system operation, which account for fully distributed system behavior, crashes of components, partitions of the underlying network, and various other problems, were developed and documented. A reasonably efficient method of coordinating the recovery of a large set of cooperating processes in a distributed or centralized environment was developed. On going activity of arbitrary members of the cooperating group is permitted by the recovery method. In addition, a suitable system architecture in which this facility can operate was designed. It preserves kernel based concepts so that the overall security of the system would be easier to demonstrate. All of this work is described in further detail later in this technical report.

During this period, we also analyzed the cost of enforcing integrity constraints in data bases by constructing analytic models for each of the possible approaches. We were able to show that run time enforcement, under reasonable assumptions, is superior to all of the other proposed methods, including enforcement at compile time. This work is reported at the 1979 ACM Sigmod Conference.

Task IV - UCLA Secure System Enhancment

A great deal of progress was made on the verification of the UCLA Secure Unix kernel during this period. A variant of the Alphard methods proposed by researchers at Carnegie Mellon University was developed for use, and successfully applied in portions of the proof effort. As a result of the substantial progress, both in developing an overall proof method as well as in its application, changes were made both in the system implementation and the abstract model which had previously been published.

The body of this technical report consists of a presentation of the substantial results in reliable distributed systems mentioned earlier under Task III.



| | |
|--|-------------------------|
| Accession For | |
| NTIS CRA&I <input checked="" type="checkbox"/> | |
| DOD TAB <input type="checkbox"/> | |
| Unpublished <input type="checkbox"/> | |
| Identification | |
| By _____ | |
| Distribution/ | |
| Availability Codes | |
| Dist | Available or Special |
| A-1 | |

BIBLIOGRAPHY

- Popek, G. J. "Correctness in Access Control," Proceedings of the ACM National Conference, Atlanta, Georgia, August 1973, pp. 236-241. Also published as Technical Report 73-1, University of California, Los Angeles, August 1973.
- Popek, G. J. and R.P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures," ACM/SIGOPS Fourth Symposium on Operating System Principles, Yorktown Heights, New York, October 15-17, 1973. Revised version published in Communications of the ACM, 17(7):412-421, July 1974. Also published as Technical Report 73-2, University of California, Los Angeles, October 1973.
- Popek, G. J. "A Principle of Kernel Design," AFIPS Conference Proceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 977-978. Also published as Technical Report 74-1, University of California, Los Angeles, May 1974.
- Popek, G. J. and C.S. Kline. "Verifiable Secure Operating System Software," AFIPS Conference Poceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 145-151. Also published as Technical Report 74-2, University of California, Los Angeles, May 1974.
- Popek, G. J. "Protection Structures," IEEE Computer, June 1974, pp. 22-23. Also published as Technical Report 74-3, University of California, Los Angeles, June 1974.
- Popek, G. J. "A Note on Secure Data Base Design," Computer Science Department, University of California, Los Angeles, Technical Report 74-4, July 1974.
- Popek, G. J. and C.S. Kline. "The Design of a Verified Protection System," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 13-14, 1974, pp. 183-196. Also published as Technical Report 74-5, University of California, Los Angeles, August 1974.
- Snuggs, M.A.L., G.J. Popek, and R.J. Peterson. "Data Base System Objectives as Design Constraints," Proceedings of the ACM National Conference, San Diego, California, November 1974, pp. 641-647. Also published as Technical Report 74-6, University of California, Los Angeles, November 1974.
- Popek, G. J. "On Data Secure Computer Networks," Proceedings of the ACM SIGCOMM/SIGARCH Workshop on Network Communications, Santa Monica, California, March 1975, pp. 59-62. Also published as Technical Report 75-1, University of California, Los Angeles, March 1975.

Popek, G. J. and C. S. Kline. "A Verifiable Protection System," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, California, April 21-23, 1975, pp. 294-304. Also published as Technical Report 75-2, University of California, Los Angeles, April 1975.

Walton, E. J. The UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, June 1975. Also published as Technical Report 75-3, University of California, Los Angeles, June 1975.

Popek, G. J. "On the Current State of Protection in Computer Systems," Eleventh IEEE Computer Society Conference: Compcon '75, Washington, D.C., September 9-11, 1975, pp. 40-41. Also published as Technical Report 75-4, University of California, Los Angeles, September 1975.

Popek, G. J. and C. S. Kline. "The PDP-11 Virtual Machine Architecture: A Case Study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975. Also published as Technical Report 75-5, University of California, Los Angeles, October 1975.

Walton, E. J. "The UCLA Pascal Translation System," January 1976, UCLA-ENG-7954 (DAHC-73-C-0368). Also published as Technical Report 76-1, University of California, Los Angeles, January 1976.

Popek, G. J. and D. A. Farber. "On Computer Security Verification," Twelfth IEEE Computer Society International Conference: Compcon '76, San Francisco, California, February 1976, pp. xxx-xxx. Also published as Technical Report 76-2, University of California, Los Angeles, February 1976.

Farber, D. A. "A Model of Program Translation," Computer Science Department, University of California, Los Angeles, Technical Report 76-3, August 1976.

Popek, G. J., J. J. Horning, B. W. Lampson, R. L. London, and J. G. Mitchell. "The Programming Language Euclid," Computer Science Department, University of California, Los Angeles, Technical Report 76-4, September 1976.

Walker, B. J. "Note on Proof of Concrete Code," Computer Science Department, University of California, Los Angeles, Technical Report 76-5, September 1976.

Abraham, S. M. A Protection Design for the UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, December 1976. Also published as Technical Report 76-6, University of California, Los Angeles, December 1976.

Popek, G. J., J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. "Notes on the Design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 28-30, 1977, ACM, Inc., New York, New York, 1977, pp. 11-18. Also published in SIGPLAN Notices, 12(3):11-18, 1977. Also published as Technical Report 77-1, University of California, Los Angeles, March 1977.

Kline, C. S. and G. J. Popek. "Encryption in Computer Network Security," Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

Kampe, M., C. S. Kline, G. J. Popek, and E. J. Walton. "The UCLA Data Secure Operating System Prototype," Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

Popek, G. J. and C. S. Kline. "Design Issues for Secure Computer Networks," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 518-546 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis). Also published as Technical Report 77-4, University of California, Los Angeles, July 1977.

Popek, G. J. and C. S. Kline. "Issues in Kernel Design," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 209-226 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis). Also published in AFIPS Conference Proceedings, Vol. 47: 1978 National Computer Conference, Anaheim, California, June 5-8, 1978, AFIPS Press, Montvale, New Jersey, 1978, pp. 1079-1086. Also published as Technical Report 77-5, University of California, Los Angeles, July 1977.

Menesce, D. A., G. J. Popek, and R. R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Systems," Computer Science Department, University of California, Los Angeles, Technical Report 77-6 (UCLA-ENG-7808), October 1977.

Popek, G. J. and C. S. Kline. "Encryption Protocols, Public Key Algorithms and Digital Signatures in Computer Networks," Proceedings of the Atlanta Conference on Fundamentals of Secure Computing, Atlanta, Georgia, October 2-5, 1977. Also published as Technical Report 77-7, University of California, Los Angeles, October 1977.

Downs, D. and G.J. Popek. "A Kernel Design for a Secure Data Base Management System," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October 6-8, 1977, IEEE, 1977, pp. 507-514. Also published as Technical Report 77-8, University of California, Los Angeles, October 1977.

Walker, B.J. "Verification of the UCLA Security Kernel: Data Defined Specifications," Computer Science Department, University of California, Los Angeles, Technical Report 77-9 (UCLA-ENG-7B09), November 1977 (Also published as a M.S. Thesis).

Kemmerer, R.A. "A Proposal for the Formal Verification of the Security Properties of the UCLA Secure UNIX Operating System Kernel," Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7B10), February 1978.

Popek, G.J. "Secure Distributed Processing Systems: Quarterly Management Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-2, April 30, 1978.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Databases," Proceedings of the 1978 ACM/SIGMOD International Conference on the Management of Data, Austin, Texas, May 30-June 2, 1978, pp. 1-14. Accepted for publication in ACM Transactions in Database Systems. Also published as Technical Report 78-3, University of California, Los Angeles, May 1978.

Popek, G.J. "Security in Network Operating Systems: A Survey," report prepared for the Institute for Computer Sciences and Technology, National Bureau of Standards, June 30, 1978. Also published as Technical Report 78-4, University of California, Los Angeles, June 1978.

Popek, G.J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-5 (UCLA-ENG-7B53), June 1978.

Badal, D.Z. and G.J. Popek. "A Proposal for Distributed Concurrency Control for Partially Replicated Distributed Database Systems," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 273-286. Also published as Technical Report 78-6, University of California, Los Angeles, August 1978.

Popek, G. J., M. Kampe, C. S. Kline, A. H. Stoughton, M. P. Urban, and E. J. Walton. "UCLA Data Secure UNIX - A Secure Operating System: Software Architecture," Computer Science Department, University of California, Los Angeles, Technical Report 78-7 (UCLA-ENG-7854), August 1978.

Menasce, D. A. and R. R. Muntz. "Locking and Deadlock Detection in Distributed Databases." Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 215-232. Also published as Technical Report 78-8, University of California, Los Angeles, August 1978. Also published in IEEE Computer Society International Computer Software and Applications Conference: Distributed Data Base Management (Tutorial), Chicago, Illinois, IEEE, New York City, New York, 1978, pp. 95-112. Accepted for publication in IEEE Transactions on Software Engineering.

Popek, G. J. and D. A. Farber. "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, 21(9):737-749, September 1978. Also published as Technical Report 78-9, University of California, Los Angeles, September 1978.

Kline, C. S. and G. J. Popek. "Public Key vs. Conventional Key Encryption," Computer Science Department, University of California, Los Angeles, Technical Report 78-10, October 1978. Submitted for publication to Proceedings of the 1979 National Computer Conference.

Popek, G. J., M. Kampe, C. S. Kline, A. H. Stoughton, M. P. Urban, and E. J. Walton. "UCLA Secure UNIX," Computer Science Department, University of California, Los Angeles, Technical Report 78-11, October 1978. Submitted for publication to Proceedings of the 1979 National Computer Conference.

Menasce, D. A., G. J. Popek, and R. R. Muntz. "Centralized and Hierarchical Locking in Distributed Databases," IEEE Computer Society International Computer Software and Applications Conference: Distributed Data Base Management (Tutorial), Chicago, Illinois, IEEE, New York City, New York, 1978, pp. 178-195. Also published as Technical Report 78-11, University of California, Los Angeles, December 1978.

Menasce, D. A. "Coordination in Distributed Systems: Concurrency, Crash Recovery and Database Synchronization," Ph.D. in Computer Science, Computer Science Department, University of California, Los Angeles, December 1978. Also published as Technical Report 78-12, University of California, Los Angeles, December 1978.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-13 (UCLA-ENG-7955), December 1978.

| TABLE OF CONTENTS | Page |
|---|-------|
| LIST OF FIGURES..... | x |
| LIST OF TABLES..... | xii |
| CHAPTER 1 INTRODUCTION..... | 1 |
| 1.1 Introduction..... | 1 |
| 1.2 Objectives..... | 2 |
| 1.3 Organization of this Dissertation..... | 3 |
| CHAPTER 2 CURRENT ISSUES IN DISTRIBUTED SYSTEMS AND DISTRIBUTED DATABASES..... | 5 |
| 2.1 Introduction..... | 5 |
| 2.2 Issues in Distributed Systems..... | 6 |
| 2.2.1 Characterization of Distributed Systems..... | 7 |
| 2.2.2 The Value of Distributed Systems. | 8 |
| 2.2.3 Major Issues in Distributed Sys- tems..... | 10 |
| 2.2.3.1 Interconnection Structures..... | 10 |
| 2.2.3.2 Error Recovery.... | 11 |
| 2.2.3.3 Network Operating Systems..... | 14 |
| 2.2.3.4 Security..... | 15 |
| 2.2.3.5 Program and Data Assignment..... | 16 |
| 2.3 Issues in Distributed Database Management Systems..... | 16 |
| 2.3.1 Data Integrity..... | 16 |
| 2.3.2 Concurrency Control..... | 17 |

| | |
|---|-----------|
| 2.3.3 Crash Recovery..... | 20 |
| 2.3.4 Synchronization Protocols..... | 22 |
| 2.4 Conclusion..... | 23 |
| CHAPTER 3 CONCURRENCY CONTROL AND DEADLOCK DETECTION IN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS. | 24 |
| 3.1 Introduction..... | 24 |
| 3.2 A Locking Protocol for Resource Coordina- tion in Distributed Databases..... | 25 |
| 3.2.1 Centralized Lock Controller Pro- tocol - Intuitive Description.... | 28 |
| 3.2.2 Lock and Release Granting Algo- rithms..... | 35 |
| 3.2.2.1 Lock Granting Algo- rithm..... | 39 |
| 3.2.2.2 Lock Releasing Al- gorithm..... | 40 |
| 3.2.2.3 State Transition Diagram..... | 41 |
| 3.2.2.4 Some Definitions and Proofs..... | 50 |
| 3.2.3 Crash Recovery..... | 51 |
| 3.2.3.1 Logical Component Recovery (LCR).... | 54 |
| 3.2.3.2 Proofs About LCR... | 62 |
| 3.2.3.3 Single Node Recovery (SNR)..... | 69 |
| 3.2.3.4 Robustness of SNR.. | 70 |
| 3.2.3.5 Logical Component Merge (LCM)..... | 71 |
| 3.2.3.6 Robustness of LCM.. | 76 |
| 3.2.4 Disjointness of the Recovery Al- gorithms..... | 77 |

| | |
|---|-----|
| 3.2.4.1 Disjointness of LCMs..... | 77 |
| 3.2.4.2 Disjointness of LCR, LCM and SNR... | 81 |
| 3.2.5 Logical Component Mutual Consistency..... | 83 |
| 3.2.6 Database Consistency..... | 84 |
| 3.2.7 Cost and Delay Analysis..... | 86 |
| 3.2.7.1 Delay Analysis.... | 88 |
| 3.2.7.2 Cost Analysis for the CLC Protocol... | 92 |
| 3.2.8 Extension..... | 94 |
| 3.3 Deadlocks in Distributed Databases..... | 99 |
| 3.3.1 Formal Model of Transaction Processing..... | 101 |
| 3.3.2 Deadlock Detection Approaches.... | 105 |
| 3.3.3 A Hierarchically Organized Locking and Deadlock Detection Protocol..... | 106 |
| 3.3.3.1 Hierarchical Protocol - A Description..... | 110 |
| 3.3.3.2 Hierarchical Protocol - Plausibility Argument..... | 114 |
| 3.3.3.3 Deadlock Resolution..... | 122 |
| 3.3.3.4 Hierarchy Establishment..... | 124 |
| 3.3.4 A Distributed Locking and Deadlock Detection Protocol..... | 125 |
| 3.3.4.1 Distributed Protocol - A Description..... | 127 |

| | |
|--|------------|
| 3.3.4.2 Distributed Proto- col - Plausibility Argument..... | 129 |
| 3.4 Related Work..... | 131 |
| 3.5 Conclusion..... | 143 |
| CHAPTER 4 A FORMAL MODEL OF CRASH RECOVERY IN COMPUTER SYSTEMS..... | 146 |
| 4.1 Introduction..... | 146 |
| 4.2 The Crash Recovery Problem..... | 147 |
| 4.3 Some Definitions and Properties..... | 150 |
| 4.4 Formal Model of Crash Recovery..... | 156 |
| 4.5 Condensed History Graph..... | 165 |
| 4.6 Operations on the Graph..... | 174 |
| 4.6.1 Adding a New Historical Version.. | 177 |
| 4.6.2 Adding an Interaction Edge and Discarding Historical Versions... | 177 |
| 4.7 Crash Set Calculation..... | 180 |
| 4.7.1 Algorithm for Crash Set Calcula- tion..... | 185 |
| 4.8 Crash Recovery of Very Large Databases.... | 191 |
| 4.9 Conclusion..... | 194 |
| CHAPTER 5 CRASH RECOVERY IN DISTRIBUTED SYSTEMS..... | 195 |
| 5.1 Introduction..... | 195 |
| 5.2 Partitioning the Graph..... | 196 |
| 5.3 Update of the Local Subgraphs..... | 198 |
| 5.3.1 Addition of a New Historical Ver- sion..... | 198 |
| 5.3.2 Addition of an Interaction Edge.. | 199 |
| 5.3.2.1 Collapsing Proto- col..... | 201 |

| | |
|---|------------|
| 5.3.2.2 Collapsing Protocol - Intuitive Description..... | 208 |
| 5.3.2.3 Collapsing Protocol - Algorithmic Description..... | 210 |
| 5.3.2.4 Collapsing Protocol - An Assertion..... | 215 |
| 5.3.3 Intentional or Accidental Loss of an Historical Version..... | 217 |
| 5.4 Distributed Crash Set Calculation..... | 218 |
| 5.5 Conclusion..... | 223 |
| CHAPTER 6 THE UCLA DISTRIBUTED SECURE SYSTEM BASE.... | 225 |
| 6.1 Introduction..... | 225 |
| 6.2 Architectural Principles..... | 226 |
| 6.2.1 Network Independence..... | 226 |
| 6.2.2 Multiple Copies..... | 227 |
| 6.2.3 Error Confinement..... | 230 |
| 6.2.4 Minimization of Recovery Relevant Data..... | 223 |
| 6.2.5 Separation of Security Relevant from Recovery Relevant Mappings.. | 234 |
| 6.2.6 Separation of Mechanisms from Policy..... | 235 |
| 6.3 Major Systems Modules..... | 237 |
| 6.3.1 Base Kernel (BK)..... | 238 |
| 6.3.1.1 Added Kernel Calls. | 239 |
| 6.3.1.2 Capability Representation.... | 240 |
| 6.3.1.3 Network Wide Inter-process Communication..... | 241 |

| | |
|---|------------|
| 6.3.2 Multiple Copy Manager (MCM)..... | 241 |
| 6.3.3 File Policy Manager (FPM)..... | 248 |
| 6.3.4 Recovery Manager (RM)..... | 250 |
| 6.4 Conclusion..... | 257 |
| CHAPTER 7 CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH..... | 259 |
| 7.1 Introduction..... | 259 |
| 7.2 Contributions to the Area of Distributed Systems..... | 259 |
| 7.2.1 Error Recovery..... | 260 |
| 7.2.2 Network Operating Systems..... | 262 |
| 7.3 Contributions to the Area of Distributed Databases..... | 263 |
| 7.4 Suggestions for Future Research..... | 264 |
| Bibliography..... | 268 |
| Appendix A: Proofs for Assertions in Chapter 3..... | 276 |
| Appendix B: Algorithm for Distributed Crash Set Calculation..... | 281 |

LIST OF FIGURES

| | Page |
|--|------|
| 2.1 Example of a Well Formed and Two-Phase Transaction..... | 19 |
| 3.1 Assured Communications Protocol..... | 38 |
| 3.2.a STD for the LC..... | 42 |
| 3.2.b STD for an LLC..... | 42 |
| 3.3 Synchronized Conversation [A,B]..... | 45 |
| 3.4 Rule to Find the Set of Global Feasible States Associated with Transaction m | 48 |
| 3.5 STDs for LCR..... | 63 |
| 3.6 State Transition Diagram for P-S Connection Establishment..... | 74 |
| 3.7 Node State Transition Diagram..... | 82 |
| 3.8 Double Time Axis Diagram for a Lock Request.... | 87 |
| 3.9 Hierarchy of Lock Controllers..... | 96 |
| 3.10 A Transaction-Wait-For Graph for a Network with Two Sites S1 and S2..... | 104 |
| 3.11 A Hierarchy of Lock Controllers..... | 108 |
| 3.12 Operation 1 in Theorem 3.10..... | 120 |
| 3.13 Operation 2 in Theorem 3.10..... | 120 |
| 3.14 Example of a Deadlock..... | 130 |
| 4.1 Time Axis Diagram..... | 149 |
| 4.2 History Graph..... | 159 |
| 4.3 Graphs G_x and G_y | 161 |
| 4.4 Condensed History Graph..... | 166 |
| 4.5 Example of Generation of Cycles and Self-Loops due to Collapsing of Nodes..... | 176 |
| 4.6 Graph $G^*(b)$ | 181 |

| | | |
|-----|---|-----|
| 4.7 | Illustration for Definitions Made in Theorem 4.7..... | 184 |
| 5.1 | Adding the Interaction Edge (x,y,t) | 202 |
| 5.2 | Example of the Node Collapsing Operation..... | 206 |
| 5.3 | Example of the Application of the Collapse Pro- tocol to the Graph of Figure 5.1..... | 216 |
| 5.4 | Global Graph G_c Distributed Among Sites S_1 , S_2 and S_3 | 220 |
| 5.5 | PSTs for the Application of the Distributed Crash Set Calculation Algorithm to the Graph of Figure 5.4..... | 222 |
| 6.1 | Complete Object-Location Mapping..... | 236 |
| 6.2 | Layered Architecture of the DSSB..... | 239 |
| 6.3 | Possible Interactions Between the FPM, MCM and the BK..... | 251 |
| B.1 | Example to Illustrate the Need of the "Freeze" Operation..... | 285 |

LIST OF TABLES

| | |
|---|-----|
| Table 1 Feasible Global States..... | 49 |
| Table A.1 Actions Taken by the Algorithm that Builds the Sets L and R..... | 279 |

ACKNOWLEDGEMENTS

I would like to express my appreciation to my doctoral committee, consisting of Professors Kirby Baker, Leonard Kleinrock, R. Clay Sprowls, Richard Muntz (Co-chairman), and Gerald Popek (Co-chairman). In particular I thank Professors Muntz and Popek for their guidance, insightful discussions, friendship and extreme dedication to this work.

Appreciation also goes to the members of the Secure Systems and Software Architecture group at UCLA for providing a stimulating research environment.

Gratitude is owed to the Conselho Nacional de Desenvolvimento Cientifico e Tecnologico (CNPq), Brazil, for their continued financial support during the three years I spent at UCLA, and also to the Advanced Research Projects Agency of the Department of Defense who supported this research under Contract Number MDA 903-77-C-0211.

Above all, I would like to thank my wife Gilda whose love and companionship made this, at times difficult, period of our lives a pleasant one.

ABSTRACT OF THE DISSERTATION

Coordination in Distributed Systems:
Concurrency, Crash Recovery and Database Synchronization

by

Daniel Alberto Menasce
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1978
Professor Gerald J. Popek, Co-chairman
Professor Richard R. Muntz, Co-chairman

The design of distributed systems requires that several aspects of resource coordination such as concurrency control, crash recovery and database synchronization be adequately treated.

A locking protocol to coordinate access to a distributed database and to maintain system consistency throughout normal and abnormal conditions is presented in this dissertation. The protocol is robust in the face of failures of any participating site and in the face of network partition-

ing. Recovery is done in such a way that maximum forward progress is achieved. The proposed protocol supports the integration of virtually any locking discipline including predicate locking methods. A cost and delay analysis of the protocol reveals that its performance does not depend on the size of the network for most topologies of interest. The protocol is formally described using state transition diagrams and a proof of its correctness is included in this work. A proposal for an extension aimed at optimizing operation of the protocol to adapt to highly skewed distributions of activity is also presented.

Two protocols for the detection of deadlocks in distributed databases - a hierarchically organized and a distributed one - are introduced in this dissertation. A graph model which depicts the state of execution of all transactions in the system is used by both protocols. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. Nevertheless, neither protocol requires that the global graph be built and maintained in order for deadlocks to be detected. In the case of the hierarchical protocol, the communications cost can be optimized if the topology of the hierarchy is appropriately chosen.

A solution to the problem of crash recovery in distributed systems is given here. A formal model of crash recovery using backward error recovery techniques is intro-

duced in this dissertation. The model is in the form of a graph, called condensed history graph, such that the branches of the graph are associated to snapshots of objects (processes, files, etc.). The graph has the following interesting properties. A specially defined cutset of the graph, called a crash set, gives the set of objects and their snapshots which have to be recovered if a crash occurs. Also, all the snapshots which belong to a directed cycle in the graph can be discarded since they can never be used in a crash recovery operation otherwise an inconsistent system state would result. Given a set of objects known to be in error, the crash set can be found by an efficient algorithm presented here. In distributed systems the condensed history graph is partitioned into several local subgraphs located at the several sites of the network. A distributed algorithm to perform crash recovery is required in this case. Such an algorithm is presented in this work.

Finally, the model of crash recovery developed in this work was used in the design of the recovery facilities of the UCLA Distributed Secure System Base. This system, the architecture of which is described here, is intended as a secure and highly reliable system base on top of which one can conveniently build distributed applications.

CHAPTER 1
INTRODUCTION

1.1 Introduction

A development which has significantly affected the computer science field over the last few years and that is of interest to this dissertation is the introduction of computer communication networks. The ARPANET [ROBE 70, KLEI 70], a successful experiment in packet switching computer communications networks, demonstrated the feasibility of interconnecting computers in a reliable and cost-effective way for the purpose of transferring data.

Today, when most of the basic problems in the design of computer networks seem to be solved, it becomes possible to consider distributed applications. Such applications, called distributed systems, should provide services or resources to a multitude of geographically dispersed users in such a way that any service or resource may be requested by the user using a site independent name. In this way, the user does not have to be concerned with the topology of the network nor with the location of the objects he needs.

Examples of the kinds of distributed systems we are considering in this dissertation include distributed database management systems and office automation systems.

This dissertation is primarily concerned with aspects of coordination of resources in distributed systems. Solutions to issues such as concurrency, crash recovery and database synchronization are given here.

1.2 - Objectives

The goals of this dissertation are:

1. to develop synchronization protocols for coordination of access to distributed databases. These protocols should be robust in the face of additional failures, preserve database consistency and exhibit reasonable performance.
2. to develop algorithms for detection of deadlocks in distributed databases. These algorithms should not require complete information about the status of all the active transactions in the system and or system resources in order for all possible deadlocks to be detected.
3. to develop a formal model of crash recovery in computer systems. This model should represent the history of each recoverable object in the system as well as the information flow between them.
4. to develop algorithms, based on the formal model

mentioned above, to perform crash recovery in centralized and distributed systems. The algorithms for distributed systems should take into account the fact that there is no central repository of information regarding the status of the various objects in the system.

5. to outline the architecture of the UCLA Distributed Secure System Base (DSSB) [MENA 78c]. The recovery facilities of the DSSB utilize the model and algorithms for crash recovery mentioned above.

1.3 - Organization of this dissertation

In chapter 2 we identify the major issues in the design of distributed systems, namely: interconnection structures, error recovery, user interface, security and program and data assignment. Also, in this chapter we discuss some of the aspects of the design of distributed database management systems such as data integrity, concurrency control, crash recovery and synchronization protocols.

Chapter 3 addresses the issue of concurrency control and deadlock handling in distributed databases. A locking protocol with adaptive centralized control and distributed recovery procedures is presented. A hierarchical extension of this protocol is also proposed. In the same chapter, we

present two deadlock detection protocols for distributed databases - a distributed one and a hierarchically organized one.

Chapter 4 is concerned with issues of crash recovery in computer systems. In particular, it addresses the following problem. Given a set of objects (e.g. processes, files, etc.), a set of snapshots for each of them and the records of information flow between them, find the set of objects and their snapshots which should be used to restore the state of the system to a consistent state, once an error in any of the (potentially interacting) objects is detected. An efficient algorithm to perform consistent state restoration in a centralized computer system is given in this chapter.

Protocols for performing consistent state restoration in distributed systems are developed and presented in chapter 5.

Finally, chapter 6 presents an outline of the architecture of the UCLA Distributed Secure System Base.

CHAPTER 2
CURRENT ISSUES IN DISTRIBUTED SYSTEMS
AND DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

2.1 - Introduction

This chapter discusses the major issues involved in the design of distributed systems and of distributed database management systems. Some examples of applications for the kind of distributed systems we are considering here are: office automation systems, message systems and distributed database management systems.

Office automation systems are systems intended to automate all the operations in an office. Such systems can be envisioned as a collection of intelligent terminals or work stations connected via a local network. A typical work station would consist of a CRT display, a keyboard, local main memory, a processor and a floppy disk. These stations would have enough processing power to carry out word processing functions such as text editing and formatting and also for running related software. Office automation systems must include some form of message processing system and some form of distributed database management system.

Message Processing Systems provide automatic message handling facilities such as: message composition, message

distribution, message filing, message retrieval, message annotation and the like. Message Processing Systems are generally a part of an office automation system.

A Distributed Database Management System provides the means through which data elements which are stored in multiple, independently operating sites of a computer network, can be accessed. Such systems must include the functional capability to locate and access the requested data and to ensure that the consistency of the database is preserved in the face of concurrent access and in the face of crashes.

2.2 - Issues in Distributed Systems

The phrase "distributed system" has been used so often to refer to fundamentally different classes of systems that its meaning has become too vague to be useful. Some examples of classes of systems, which have been labeled as distributed systems are: packet switching computer communications networks such as the ARPANET [ROBE 70], network operating systems such as the RSEXEC [THOM 73] and the National Software Works (NSW) [CROC 75], tightly coupled microprocessor systems such as the Cm* [SWAN 77] and even collections of intelligent terminals (e.g. terminals with some local editing and formatting capability) connected to a mainframe in a star-like configuration.

2.2.1 - Characterization of Distributed Systems

To narrow down the characterization of distributed systems we will use a particular definition, originally given in [ENSL 78], which encompasses the motivations laid out in the previous section.

"A distributed system is a collection of physically distributed elements (e.g. processors, terminals, etc.) interacting through a computer communications network in a cooperative fashion. It is also required that there be a high level operating system that unifies and integrates the control of the distributed elements and that provides system transparency to users. It should be possible for the various system resources to be dynamically assigned to tasks and there should be multiple instances of these resources."

Some comments about the above definition are in order. The communications network assumes that information flow between two sites takes place according to a two-party protocol. The elements that compose the system should interact in a cooperative fashion at the logical level and at a physical level. This interaction should preserve the autonomy of each of the interacting elements. The high level operating system should allow users to refer to the services they need with no concern over the identity and location of the

server providing it.

Therefore, for our purposes, computer networks, multiprocessors and intelligent terminals will not be considered among the distributed systems addressed in this thesis. Computer networks do not exhibit the system transparency required by the definition. Multiprocessors by themselves, with no high level operating system cannot be considered a distributed system. Intelligent terminals among other things, do not meet the requirement that interaction between system elements be cooperative.

2.2.2 - The Value of Distributed Systems

One of the reasons for having a distributed system is to achieve resource sharing. One would like to be able to share specialized software or hardware resources, such as image processing programs, word processors or number crunching machines.

Distributed systems, if properly designed, have the potential of exhibiting high reliability and high availability over centralized systems. This is only possible if enough redundancy is available, if efficient and correct error recovery mechanisms are implemented, and if the architecture assures that errors do not propagate.

Another motivation for having distributed systems is the potential for automatic load sharing, adaptability to changes in the work load and good response to transient overloads. In order for the just mentioned characteristics to be present the system must have the ability to dynamically move executing tasks to other sites.

With the advent of large scale integration (LSI), hardware costs have declined in such a way that today the cost of a mainframe is an order of magnitude greater than the one composed of a collection of minicomputers , which have collectively the same computational power as the mainframe. In other words, distributed systems exhibit less cost for more computational power.

The decrease in hardware costs made it feasible for small departments within an enterprise to have their own small computers [SALT 78]. There is need for these machines to be connected together so that operations that span departments (e.g. weekly financial reports of the enterprise) can be automated. This arrangement reflects the organization structure and satisfies its computational needs in a rather convenient manner. Related advantages of distributed systems are: ease of expansion and easy adaptation to new functions.

2.2.3 - Major Issues in Distributed Systems

Let us now identify the major issues in the design of distributed systems.

2.2.3.1 - Interconnection Structures

The different machines in a distributed system can be connected in several distinct ways. Among the possibilities one should distinguish between local and geographically distributed networks. Networks which interconnect computers located in a reasonably small area - of the order of a few square miles - such as an office building or even a room are called local networks. For such short distances, one can use communication lines which exhibit high bandwidth and low delay. With current technology, high bandwidth wires cannot be used for distances over one mile or so. Two examples of local networks are the Distributed Computing System (DCS) built at the University of California, Irvine [FARB 72c] and the ETHERNET built at the Xerox Palo Alto Research Center [METC 76]. DCS has a ring topology - every node is connected to two others. The ETHERNET is a broadcast type of network in which all the nodes are connected to a transmission medium, called the "ether", which is pair of twisted cables. Messages sent on the ether can be received by all the nodes - this is called a broadcast type of transmission. Messages can be destroyed due to collisions with other messages

transmitted almost simultaneously. Collision detection and retransmission protocols are required.

If the machines on the network are spread through a large geographical area, we have a geographically distributed network. Perhaps the best known example of such a network is the ARPANET [ROBE 70]. Private or leased telephone lines and/or satellite links are used to connect computers in distributed networks. These connections have low bandwidth and high delay characteristics.

The kind of interconnection structure, i.e. local or geographically distributed, certainly impacts considerably the design of a distributed system. As an example, the design of the DSSB described in chapter 6 of this dissertation, utilizes the fact that fat wires are available to implement a network wide page faulting mechanism.

2.2.3.2 - Error Recovery

As we pointed out in section 2.2.2, one of the main objectives of a distributed system is to have high reliability and high availability. The potential for achieving this goal is present in distributed systems. However, in order for this potential to materialize, it is necessary that appropriate error recovery facilities be built into the system. These facilities should be mostly automatic since the user has little or no control over the current location of

its processes and files given that system transparency is one of the desirable properties of a distributed system. Also, in general we try to insulate users from problems not directly related to getting their task done.

There are basically two strategies for error recovery; namely, backward error recovery and forward error recovery. In the former, the system must be backed up to a previously saved snapshots. The collection of snapshots of the objects backed up (e.g. files, processes) must restore the system to a globally consistent state, i.e., a state which the system might have reached through normal operation, starting at a consistent state. The latter strategy requires that the state of the damaged objects be corrected - with the aid of some redundancy - and that operations be restarted. The applicability of forward error recovery techniques is very much dependent on the nature of the error itself and of its consequences.

While error recovery is not a unique characteristic of centralized systems, it becomes harder and crucial in distributed ones. The main reason is the time delay inherent to computer networks combined with the fact that status information about the several objects in the system, is distributed and not centralized. In this dissertation we address formally the issue of crash recovery in computer systems (see [MENA 79]) and we develop algorithms to carry out

error recovery in distributed systems.

2.2.3.3 - Network Operating Systems

As we mentioned in section 2.2.1, a high level operating system is a necessary element of a distributed system. This operating system will be called network operating system (NOS). An NOS should provide to the user a transparent view of the system in the sense of allowing services to be requested from the system by site independent names and not from servers whose locations must be known by the user. Therefore, the different resources of the system should be controlled and made accessible to the user by the NOS in a convenient and cost-effective way. One requirement of an NOS is that all of its functions, such as resource allocation, scheduling, synchronization, be decentralized.

A network operating system should have the following two components:

1. A network wide interprocess communication mechanism. This mechanism coupled with a network wide process name space makes the network transparent as far as interprocess communication is concerned.
2. A distributed file system together with a network wide file name space. Such a mechanism makes the

network transparent as far as file usage is concerned.

Two network operating systems have been developed for a collection of nodes in the ARPANET. One of them, the RSEXEC [THOM 73], provides a distributed file system which makes all files, regardless of location, uniformly accessible. The other is the National Software Works (NSW) [CROC 75] which is designed to provide the users of the system with a uniform way of accessing a variety of software tools regardless of their location in the network. Unlike RSEXEC, file operations are logically centralized in NSW due to the existence of a central catalog used to resolve file references.

A third example of a distributed operating system is the one developed for the Distributed Computing System (DCS) [FARB 72 and ROWE 75]. An interesting feature of this system is its resource allocation scheme which is based on a bidding strategy. A process which requires a service bargains with the processes which can supply it. The requesting process then picks up the bidder which has indicated a minimal amount of induced overhead.

2.2.3.4 - Security

The issue of security in centralized computer systems has gained considerable attention from the research community over the past few years. Among the leading projects in the area is the UCLA Data Secure Unix Operating System [KAMP 77]. It is a kernel based operating system. All the security relevant functions of the operating have been isolated in the kernel which has been designed to be amenable to validation [POPE 74]. The kernel is currently being verified for its security properties [KEMM 79 and POPE 78b].

Decentralization adds another dimension to the problem of security in computer systems since communications lines and switching processors cannot be assumed to be secure. As pointed out by Popek and Kline in [POPE 78a] the environment presents several threats such as the tapping of communication lines, introduction of spurious traffic, retransmission of previously transmitted genuine traffic or breaking of physical lines.

The only general solution to the above problems is the use of some form of encryption. Related issues are: the placement of the endpoints of the encryption pairs, key distribution, confinement and authentication.

2.2.3.5 - Program and Data Assignment

In order to adapt to changes in the work load and to adapt to failures, a distributed system must have the capability to move programs and files in a dynamic way. This fact raises the issue of finding optimal strategies to allocate data and files to processors in order to minimize cost.

2.3 - Issues in Distributed Database Management Systems

This section highlights the major issues in distributed database management systems.

2.3.1 - Data Integrity

A database is an information model of a real world environment and as such it should reflect the properties of this environment. In general, the database integrity is said to be violated if there is an inconsistency between it and the real world it models. Two distinct types of consistency notions have to be considered in DDBs, namely: internal consistency and mutual consistency.

The notion of internal consistency applies to distributed databases as well as to centralized ones. A database is said to be in an internally consistent state if it satisfies a set of predefined assertions or consistency constraints. For instance, in an airline reservation sys-

tem, a consistency constraint may be that the number of reserved seats does not exceed the capacity of the plane. In a banking system we might require that the sum of all the balances of customer's accounts and assets be equal to zero.

The notion of mutual consistency applies only to DDBs with multiple copies. As defined in [THOM 76], a set of multiple copies of a DDB is said to be mutually consistent if after all activity ceases they all converge to the same value.

Eswaran et al. introduced, in [ESWA 76], the notion of transaction as the unit of consistency. A transaction, which is a set of actions of the form: read, write, lock or unlock, takes the DB from a consistent state into another consistent state. Even if transactions are programmed in such a way that they individually preserve the consistency constraints of the DB, consistency can be compromised by the two following factors: concurrent access to the database and the occurrence of crashes. These issues will be discussed in the following sections.

2.3.2 - Concurrency Control

If each transaction by itself preserves the consistency constraints and if transactions are executed one after the other, i.e., in a serial way, the database internal consistency is preserved. However, one may want to interleave

the execution of the actions of several transactions in order to increase system concurrency. This interleaving cannot be arbitrary otherwise transactions could be presented with an inconsistent view of the database. To understand the issue consider the following banking transaction.

```
TRANSFER $5 FROM SAVINGS ACCT # XXX  
INTO CHECKING ACCT # YYY.
```

After the five dollars have been withdrawn from the savings account but before they are credited into the checking account, the database is temporarily in an internally inconsistent state. This temporary inconsistency can be tolerated as long as it cannot be "seen" by any other transaction, i.e. another transaction cannot read the data which is inconsistent. We need therefore, a concurrency control mechanism to control the way in which the actions of several transactions are interleaved in order to preserve DB consistency.

A particular sequence of the actions of several transactions is called a schedule. A serial schedule is a sequence of actions such that between two actions of the same transactions there is no action of any other transaction. It is clear that serial schedules preserve database consistency. A schedule is said to be serializable if it produces the same result as some serial schedule.

As shown in [ESWA 76] transactions should satisfy two conditions in order for any schedule to be serializable, namely they should be well formed and two-phase. A transaction is said to be well formed if all the data items it accesses are locked in the appropriate mode by the transaction. For instance, before transaction T is able to write into data item **x**, this data item must be locked in exclusive mode by T. A transaction is said to be two-phase if it is not allowed to acquire any further locks after it has released the first lock. Therefore, two phase transaction have a growing phase during which they are allowed to acquire new resources and a shrinking phase which starts when the first release resource request is issued and ends with the transaction. Figure 2.1 shows an example of a well

```
LOCK    A
READ   A
LOCK    B
READ   B
UPDATE B
UNLOCK B
UPDATE A
UNLOCK A
```

FIGURE 2.1 - Example of a well formed
and two-phase transaction.

formed and two phase transaction.

An undesirable side-effect of the need of locking resources is that of a deadlock. A deadlock is a situation in which a transaction cannot make not make any further progress because it needs a resource being held by any other

transaction which is itself waiting for a resource held by the first transaction. This mutual blocking may involve more than two transactions in a cyclic pattern. For example, consider three transactions T1, T2 and T3 and three resources a, b and c. Assume that T1 has resource a locked in exclusive mode and is waiting to acquire resource b which is held by T2 in exclusive mode. Assume that T2 is waiting for resource c which is held by T3 which is waiting for resource a. Therefore, none of the transactions is able to continue and therefore none of them is able to release the resource which is needed by the others.

There are several alternatives to handling deadlocks in DDBs, namely: deadlock avoidance, deadlock prevention, deadlock detection and resolution and transaction timeout, backout and restart. The advantages and disadvantages of the several alternatives are discussed in chapter 3 of this dissertation. Deadlock detection protocols (see [MENA 78b]) are also presented there.

2.3.3 - Crash Recovery

As we saw in the previous section, temporary inconsistencies in the DB can be generated after some of the updates of a transaction but not all of them have been applied. If a crash occurs at this point, the database will be left in an inconsistent state. In order to remedy this

situation, transactions should be atomic in the sense that either all of its updates are done or none of them is. This is called the all-or-none property. Lampson and Sturgis [LAMP 76] suggested a mechanism which makes transactions atomic. Their scheme is based on two important notions, namely the notion of stable storage and that of intentions list. Stable storage is defined as being a secondary storage in which writing a page is an atomic operation in the sense that either the whole page is modified or it is not modified at all. Stable storage can be built from normal secondary storage by associating to each logical page two physical pages. Each write to a logical page is then implemented as consecutive writes of the two physical pages associated with the logical page. Error detection bits are added to the physical pages so that one can tell, in case of crashes, how far the writing of the two pages has gone.

An intentions list is a list of actions which if carried out completely posts all the updates of a transaction into the DB. Such a list must have the property that the effect of partially executing the intentions list any number of times followed by its complete execution is the same as if the list were executed only once.

The scheme proposed in [LAMP 76] can now be summarized as follows. Intentions lists for a transaction are stored in stable storage. Once the intentions list is completely

written, one can start carrying it out. If a crash occurs within this period we only have to start to carry out the intentions list from the beginning once again. After the intentions list is written, crash recovery is guaranteed. In a distributed database management system, there is a process which coordinates the updates of a given transaction. This coordinator requests that all intentions lists be written at all the participating sites before it requests that anyone of them starts to carry them out. This protocol was also described by Gray in [GRAY 78] and called the two-phase commit protocol.

2.3.4 - Synchronization Protocols

Synchronization protocols are the set of rules which coordinate the access to a DDB. These protocols should deal with multiple copies, deadlocks, DB consistency and crash recovery.

Several such protocols have been suggested in the literature [ALSB 76, BADA 78, ELLI 77a, ELLI 77b, LELA 78, ROSE 77, MENA 78a, STON 78, and ROTH 77]. In chapter 3 of this dissertation we list the desirable properties of synchronization protocols and then we describe a protocol which satisfies the desired properties.

2.4 - Conclusion

A characterization of distributed systems was given in this chapter. The major issues in the design of such systems are: interconnection structures, error recovery, network operating systems, security and program and data assignment.

The major issues in distributed database management systems, as identified in this chapter, are data integrity, concurrency control, crash recovery and synchronization protocols.

CHAPTER 3
CONCURRENCY CONTROL AND DEADLOCK HANDLING
IN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

3.1 - Introduction

This chapter is concerned with issues of concurrency control and deadlocks in distributed database management systems. A locking protocol for coordination of resources and the maintenance of database consistency throughout normal and abnormal conditions is presented here. The protocol has centralized control and distributed recovery procedures. In this protocol, deadlocks are handled by the centralized controller.

Two other locking and deadlock detection protocols for distributed databases are presented in this chapter - one is hierarchically organized and the other is distributed. A graph model which depicts the state of execution of all transactions in the system is used by both protocols. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. Nevertheless, neither protocol requires that the global graph be built and maintained in order for deadlocks to be detected. In the case of the hierarchical protocol, the communications cost can be optimized if the topology of the hierarchy is appropriately chosen.

The chapter concludes with a discussion of related work.

3.2 - A Locking Protocol for Resource Coordination in Distributed Databases

Coordination of resources in a distributed environment exhibits additional complexity over resource coordination in centralized environments due to:

1. possibility of crashes of participating sites and or communication links. Occurrence of such failures can render the database inconsistent if not appropriately handled by the coordination algorithm.
2. network partitioning: in general, it is not possible to distinguish between messages which could not be delivered due to a crash of the recipient site and undelivered messages due to network partitioning. Therefore, network partitioning in the more general sense considered here is not simply a matter of proper network topology design. It turns out that detection of network partitioning can only occur at network reconnection time.
3. inherent communication delay: the time to get a message through a computer communication network

may be arbitrarily long, although finite. Therefore any proposed solution should operate correctly regardless of the delay experienced by any message, and in general should be efficient.

A protocol to coordinate concurrent access to a distributed database using locking is presented in this section. The algorithm has as its core a centralized locking protocol with distributed recovery procedures. A centralized controller with local appendages at each site coordinates all resource control, with requests initiated by application programs at any site. Recovery is broken down into three disjoint mechanisms; for single node recovery, merge of partitions and reconstruction of the centralized controller and tables.

Among the properties of the proposed protocol we have:

- a. robustness in the face of crashes of any participating site, as well as communication failures, is provided. The protocol can recover from any number of failures which occur either during normal operation or during any of the three recovery processes. Recovery is done in such a way that maximum forward progress is achieved.
- b. deadlock prevention and or detection methods can

be easily integrated given the centralized control characteristic of the proposed algorithm.

- c. straightforward integration of predicate locking methods [ESWA 76] is permitted. Value dependent lock specification at the logical level is necessary to avoid the problems of "phantom tuples" discussed by Eswaran et al [ESWA 76]. Other locking disciplines may also be easily supported.
- d. continued local operation in the face of network partitioning is supported. The locking algorithm operates, and operates correctly, when the network is partitioned, either intentionally or by failure of communication lines. Each partition is able to continue with work local to it, and operation merges gracefully when the partitions are reconnected.
- e. performance of the algorithm does not degrade operations. It is shown in a later section of this chapter that for many topologies of interest, the delay introduced by the protocol is not a direct function of the size of the network. The communication cost is shown to grow in a relatively slow, linear fashion with the number of sites participating in a given transaction.

f. the correct operation of the protocol in the face of the failures mentioned before can be proven in a straightforward way.

The protocol presented in this section is described in an intuitive manner in the next section, followed by a more detailed description in the two subsequent ones. An algorithmic specification of this locking protocol can be found in [MENA 77]. An informal proof of the correctness of the algorithm is presented here. The proof is decomposed into five major parts, one for normal operation, three for the recovery phases, and a last part that shows the parts actually can be proved disjointly.

A proposal for an extension aimed at optimizing operation of the algorithm to adapt to highly skewed distributions of activity is presented here. The extension applies nicely to interconnected computer networks.

3.2.1 - Centralized Lock Controller Protocol - Intuitive Description

The database we are considering here is distributed among n nodes of a computer network, numbered from 1 to n . We assume that the network protocols are such that a copy of a message is kept by its sender until an acknowledgment for it is received. In other words, there are no lost messages.

However, messages may have to be retransmitted many times until they get through the net. An implication of these assumptions is that messages may be delayed by an arbitrary but finite amount of time. We also assume that messages from a source site A are delivered by the network protocols to a destination site B in the same order they were generated. However, we make no assumptions about the order in which messages from two distinct sources are received by a third one. We require that the network routing procedures be such that every pair of nodes can communicate with each other if the necessary physical connection is available.

User interaction with the database is done through application programs, APs, which communicate with the Data Base Management processes. Of those processes, two are of interest for this locking protocol: the centralized lock controller or simply lock controller and the local lock controller.

As a first approximation assume that there is only one lock controller or LC for the entire network. This process is responsible among other things for examining lock and lock release requests from the APs, and deciding whether they should be granted or not. For this purpose, the LC maintains a table called the LOCK table, which is a set of all the active locks. Each entry in this table is a 3-tuple of the form (H,T,P) where H is a unique host identification,

T is a unique database transaction identifier within each site and P is a description of the logical portion of the database to be locked as well as the lock mode (e.g., read, write, etc.). In a relational database, the lock specification may for example be a predicate lock as described by Eswaran et al [ESWA 76]. Note that the pair (H,T) is a globally unique transaction identifier.

At every site, except for the one where the LC is located, there is a local lock controller or LLC. Those processes are responsible for maintaining a local copy of the relevant portion of the LOCK table. The relevant portion of the LOCK table is the set of entries which contain locks which refer to data stored locally. Any LLC may become the lock controller whenever there is a crash in the system which makes the LC unavailable. The recovery process is explained later in detail. Each time a transaction takes an action the local copy of the LOCK table is examined to determine whether the action can be performed or not. Therefore, there are two reasons for keeping a local copy of the LOCK table, namely: resilience to failures and local action checking.

It is convenient at this point to introduce the notion of logical partition or logical component, as opposed to that of a physical component. A physical component is a maximal subnetwork such that every pair of sites in the com-

ponent can communicate with one another. It can be readily seen that the composition of a physical component is not under the control of the locking protocol, since nodes and communication links fail independently of the protocol operation. Such a lack of control could make the operation of the protocol, in the face of crashes, rather complex. The concept of logical component is introduced to give the protocol independence from unexpected changes in the composition of each physical component. To this end, each LC keeps a list of sites which he thinks are still up, called the up list. A logical component is defined as being the subnetwork indicated by the nodes which are in the up list. This list may lag behind the list of sites which are actually up. Independence from the composition of physical components is thus achieved by controlling the way by which the latter list maps into the former, in a way which is explained later in this chapter.

Since one of our stated goals is to allow local operations to continue in face of network partitioning and to allow partitions to merge gracefully, it is necessary for each partition to have its own LC. There is one LC for each logical component.

The operation of the locking protocol under no crash conditions can be intuitively explained as follows. The LC receives lock and lock release requests from the application

programs. The LC then assigns a sequence number to the request. These sequence numbers are taken from a monotonically increasing sequence of numbers and will be used for the purposes of crash recovery as will become clear in a later section. Each request (including its sequence number) is then sent to all relevant LLCs in the component. An LLC is relevant with respect to a request if its site stores data addressed by the request in question. The request is stored in a pending list at each LLC site and an acknowledgment is sent back to the LC. After the acknowledgment from all relevant sites in the component is received (excluding those which crashed in the meantime) a confirmation for the request is sent by the LC to all LLCs causing the request to be deleted from the pending list and appended to the LOCK table.

A lock request may be rejected by a LC if it conflicts with other locks in the LOCK table or in the pending list or if the request is not local to the component. We assume that the LC is able to determine for each lock, P, the set, LOC(P), of sites where the data to be locked are stored. Thus, a lock P is said to be local if LOC(P) is contained in the up list for the component. The set LOC(P) can be determined by the LC by checking some catalogs. The organization of those catalogs is not relevant here; see [CHU 76] and [STON 76] for discussions of that subject.

Every time that a site or a set of sites drop out of the up list, all the transactions which had at least one lock which became not local will be aborted or backed up. All the locks held by these transactions are released. In this way complete locality of operations is enforced by the CLC protocol.

If the LC crashes or becomes unavailable a recovery mechanism called Logical Component Recovery (LCR) takes place. As soon as an LC-crash is detected by any process engaged in a conversation or exchange of messages with the lock controller, a new process is nominated to be the new LC. There is a globally known circular ordering of the sites from which the nominee is selected. If the nominee is up it accepts the nomination by sending a message which circulates through all the sites in the component. The purpose of this message is also to collect all the requests which have been received by all the relevant sites but which are still in the pending list for at least one of these sites. Those requests will be incorporated into the LOCK table at every site in a subsequent phase of the recovery process. In summary, the LCR mechanism amounts to electing a new LC for the component and updating all the LOCK tables appropriately before normal operation is resumed. Various race conditions are dealt with by the details of the recovery protocol.

It is the responsibility of each LC to periodically monitor the connection between it and a node not in its up list. If a physical connection between two previously logically disconnected component is detected, a Logical Component Merge (LCM) mechanism is started. LCM is always done pairwise between components and in this process the LC of one of the components plays an active role while the other plays a passive one. The first phase of LCM is composed of an interconnection protocol by which two LCs are logically connected in such a way that one of them is designated active and the other passive. This protocol also enforces the pairwise merge condition and is shown to be deadlock free. After a logical connection has been established both LCs clear all outstanding requests and reject further ones. In the subsequent phase, the union of the LOCK tables of the two components is made and the new LOCK table is sent to all the sites in both components in the form of a message which circulates through them. This message signals the completion of the merge. The active LC becomes the lock controller for the new logical component.

When a site which was down recovers, it is made active by the Single Node Recovery (SNR) mechanism which basically amounts to the acquisition by that site of a new copy of the relevant portion of the LOCK table.

The three recovery mechanisms described above do not interact with each other, as will be shown later. This property is important because it allows us to decompose the correctness proofs into a proof of disjointness and then proofs for each recovery procedure separately.

The recovery mechanisms will be shown to be robust in the face of additional failures. In order to achieve this goal, each mechanism is designed in such a way that a partial execution of any of the recovery algorithms does not destroy any of the properties we want to prove about them.

It is important to emphasize at this point that, since all the lock requests are examined by a centralized lock controller in one logical component, locks granted by an LC do not conflict with one another. This fact enables us to consider the operation of the algorithm for normal operation and for recovery as if there were only one lock per logical component. The reader is encouraged to keep this in mind as he reads through this section.

3.2.2 - Lock and Release Granting Algorithms

This section describes informally the algorithms used to grant new locks and to release existing ones. One would like those algorithms to have the property that a lock is either granted or released if and only if it is known to all the relevant sites. The basic structure of both algorithms

can be abstracted in what we call the Assured Communication Protocol (ACP) which exhibits the desired property outlined below.

Let there be a sender S, who wishes to send a message M, originated at an external source ES, to n destinations D₁, D₂, ..., D_n. Each site i keeps two message buffers: temp_buffer(i) and final_buffer(i). ACP is such that message M will only be in final_buffer(S) if M is either in temp_buffer(D_i) or final_buffer(D_i) for all destinations D_i. ACP can be described by the following set of rules:

1. S receives a "MESSAGE REQUEST" or MR message from ES and broadcasts an "ACCEPT MESSAGE" or AM message, which contains M, to all D_i's, i=1,...,n. The message M is placed in temp_buffer(S).
2. When an AM message is received by a destination D_i, the message M is placed in temp_buffer(D_i) and a "MESSAGE ACCEPTED" or MA message is sent back to S.
3. When all the MA messages have been received by S, M is moved to final_buffer(S) and removed from temp_buffer(S) and a "CONFIRM MESSAGE" or CM message is broadcast to all destinations.
4. The receipt of a CM message at destination D_i

causes M to be moved into final_buffer(D_i) and removed from temp_buffer(D_i).

A variant to this protocol, called a two-phase commit protocol, is described in [GRAY 78] and [LAMP 76]. The two-phase commit protocol has an additional acknowledgment message at the end from each destination to the sender. However, we show here that this additional message is unnecessary. Therefore, the ACP protocol has 33 percent less messages than the two-phase commit protocol. The Assured Communications Protocol is illustrated in Figure 3.1. In this figure, the horizontal arrows are labeled with message names. There is one vertical line corresponding to the sender S and another corresponding to destination D_i. Two graphical notations were used in this picture. Namely, the diverging arrows shown in point A of the vertical axis for the sender S indicate that the message in question is being broadcast to every one of the destination sites. The converging arrows shown in point B of the same vertical axis indicate that the sender must wait until it receives all the messages from all the sites which are up. Previously up sites may be pronounced down by the underlying network protocols after timeout and retransmission took place several times.

Several other details are also worth keeping in mind. As mentioned before, each LC keeps a list of the sites in

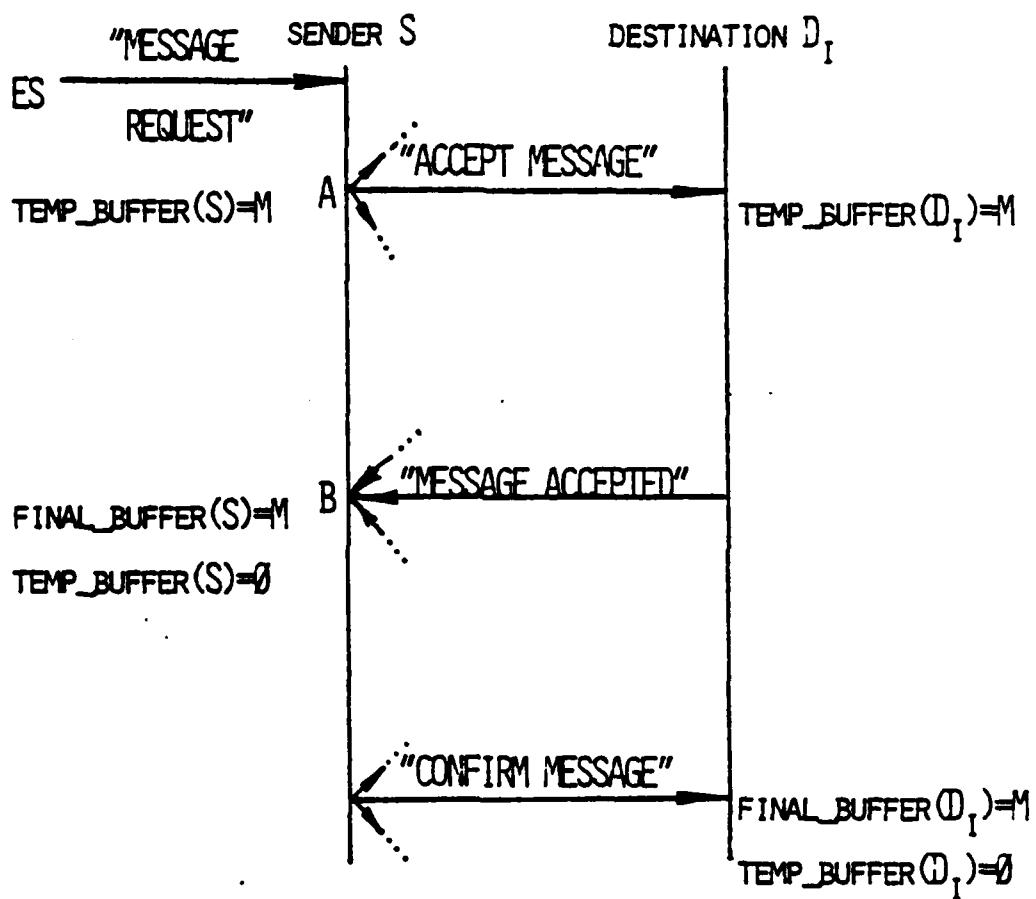


Figure 3.1 - Assured Communications Protocol (ACP).

the component which are up. A node i is removed from this list by the LC each time that the underlying network protocols fail to deliver a message to site i (after timeout and retransmission occurred a certain number of times). An up list is also modified by the execution of any of the three recovery mechanisms. A copy of the up list is also kept by each LLC. Every update to the up list by the LC is transmitted to all LLCs in the component. Note that no additional message traffic is generated by those updates since they can "piggyback" on other messages. The reason for keeping local copies of the up list is merely a matter of performance, since the up list determines to some extent the set of nodes which should participate in the LCR or LCM recovery mechanisms, as will be seen later. Also, every time that a change in the up list causes certain locks not to be local any more, all non-local locks are released and the affected transactions aborted.

3.2.2.1 - Lock Granting Algorithm

Application programs issue lock requests by sending a "LOCK REQUEST" or LR message to the LC. This message contains the lock or 3-tuple which the user would like to be entered in the LOCK table. The LC decides whether the lock can be granted or not. If the requested lock conflicts with other active locks a scheduling decision must be taken by the LC as to whether to preempt any transaction or to make

the requester wait. That decision is not the concern of this section. If there are no conflicts and the lock is local to the component the LC must notify every relevant LLC in its component that a new entry should be appended to their LOCK tables. Actually, instead of inserting the lock directly into the LOCK table, an LLC appends it to a list of pending lock requests, called an L-list. The reason for this is to prevent copies of the LOCK table from becoming inconsistent if the LC crashes.

The basic structure of the Lock Granting and Lock Releasing algorithms is the same as that of the ACP protocol, where AP, LC, LLCi and LOCK table correspond to ES, S, Di and final_buffer in ACP, respectively. Also, the message M in ACP should be considered as a lock request for the Lock Granting algorithm and as a release request for the Lock Releasing one. For the Lock Granting Algorithm, in particular, temp_buffer corresponds to an L-list.

3.2.2.2 - Lock Releasing Algorithm

A similar procedure is followed when an AP issues a lock release request, by sending to the LC a "RELEASE REQUEST" or RL message. Each site keeps a list of pending release requests or an R-list for the same reasons we introduced the L-list. The R-list corresponds to temp_buffer in the ACP protocol.

3.2.2.3 - State Transition Diagram

The operation of the Lock Granting and Lock Releasing algorithms can be described by a pair of interacting graphs or state transition diagrams as shown in figure 3.2. This technique was introduced by Zafiropulo in [ZAFI 77a] as a tool for modelling computer communications network protocols. There is one state transition diagram (STD) for the LC (figure 3.2.a) and one for a local lock controller LLC_i (figure 3.2.b). State transitions are triggered by events such as message transmission or reception. Each STD has a quiescent or initial state. As already mentioned in section 3.2.1, the LC does not grant conflicting locks, so that we can consider the operation of the protocol as if there were only one lock request. Let x be such a lock. It is useful to associate with each state s in the STD a 3-tuple (a, b, c) where a , b and c are binary variables which indicate whether the lock x is in the LOCK table, in the L-list or the R-list respectively when the protocol is in state s . For instance, the tuple $(1, 0, 1)$ indicates that the lock x is in the LOCK table and in the R-list. Labels on the state transition arcs represent conditions or events upon which the transition takes place. These conditions indicate either message transmission or reception. The following abbreviations for message names were used in the diagram of figure 3.2:

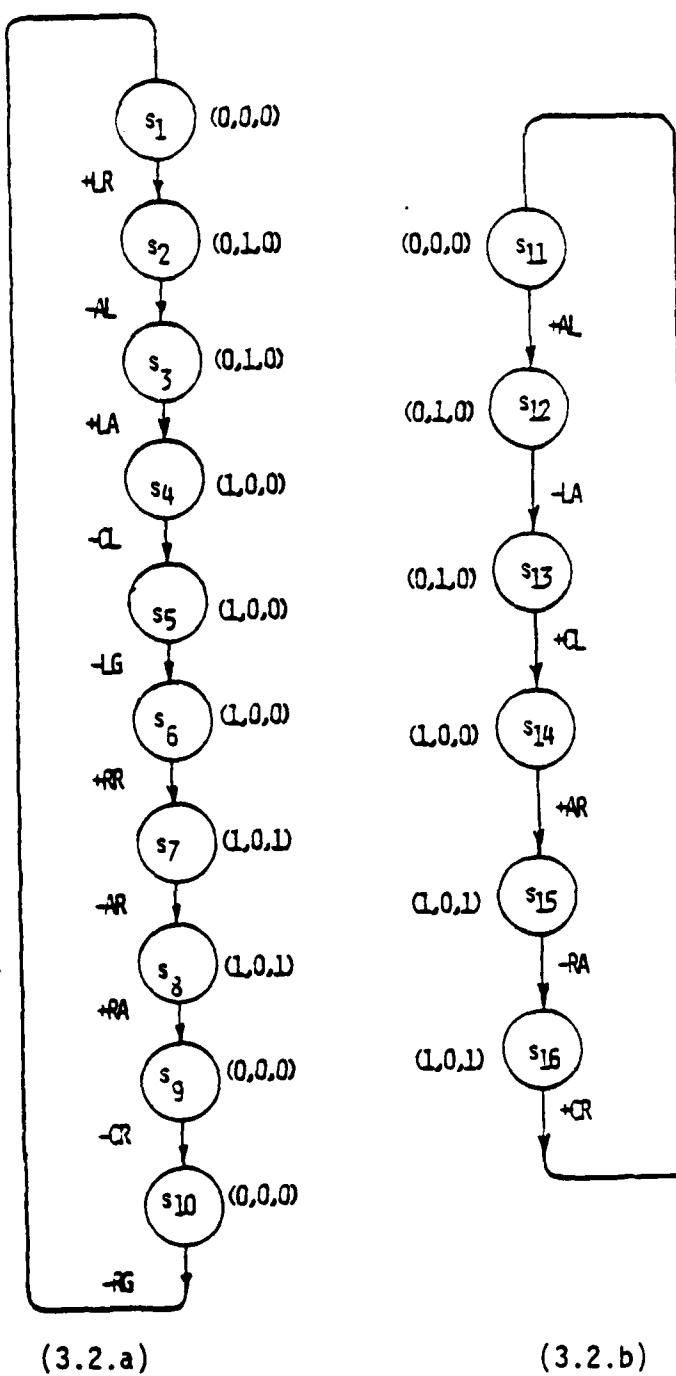


Figure 3.2 - STD for the LC (figure 3.2.a)
and for an LLC (figure 3.2.b).

From AP to LC:

LR: LOCK REQUEST
RR: RELEASE REQUEST

From LC to AP:

LG: LOCK GRANTED
RG: RELEASE GRANTED

Among LC and LLCs:

AL: ACCEPT LOCK
LA: LOCK ACCEPTED
CL: CONFIRM LOCK
AR: ACCEPT RELEASE
RA: RELEASE ACCEPTED
CR: CONFIRM RELEASE

There are state transitions in one STD which have a companion in the other STD, i.e. transmission of a message in one of them and reception of the same message in the other. We label the transitions with "signed" message names. A positive sign represents a message reception and a negative one indicates a message transmission.

Let a transition cycle be a path beginning and ending in the initial state. Let a conversation [A,B] be a pair of transition cycles A and B such that each of them belong to

different STDs. Let us define an event sequence of a transition cycle A as the sequence of labels of the transitions in A which correspond to internal* events only. A conversation [A,B] is said to be synchronized if the event sequence of A is equal to the event sequence of B except for the signs in the events which are reversed.

From the description of the protocol one can easily draw the STDs in figure 3.2. Actually an STD may be considered as a protocol specification. Let us define a global state S for a set of k STDs, STD₁ through STD_k, as a k-tuple (s₁, s₂, ..., s_k) where s_i, for i ∈ {1, ..., k}, is the 3-tuple associated with state s_i in STD_i. A global state is said to be feasible if the individual component states for each STD can coexist.

If all the conversations of a protocol are synchronized then the task of finding the set of all global feasible states is fairly straightforward. Consider the synchronized conversation [A,B] shown in figure 3.3 below.

Let -m be a condition on the transition cycle A which triggers the transition from state s[a,i] into state s[a,i+1]. Also, let +m be the companion condition in the transition cycle B which triggers the transition from state

* An event is internal if it does not represent an external action such as an exchange of messages with an application program.

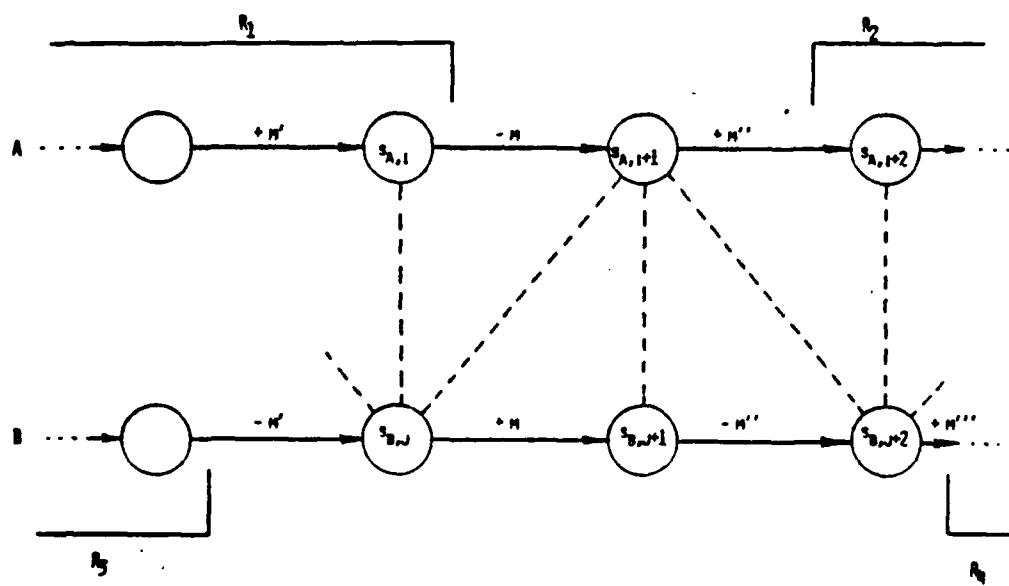


Figure 3.3 - Synchronized Conversation [A,B].
The dashed lines indicate global feasible states.

$s[b,j]$ into state $s[b,j+1]$. R_1 is the set of states in A which precede* $s[a,i+1]$. R_2, R_3 and R_4 have a similar interpretation (see figure 3.3). Let us consider first a state which is reached by a transition labeled by a message transmission, say $s[a,i+1]$ for instance. The set of possible global states which include $s[a,i+1]$ are:

1. $\{(s[a,i+1],x) \mid x \in R_3\}$: these states are not feasible because message m' was not yet sent but already received.
2. $\{(s[a,i+1],x) \mid x \in R_4\}$: these states are not feasible because m''' was not yet sent but already received.
3. states $(s[a,i+1],s[b,j]), (s[a,i+1],s[b,j+1])$ and $(s[a,i+1],s[2k])$ are feasible.

Let us consider now a state which is reached by a transition labeled by a message reception, $s[b,j+1]$ for instance. The set of all the global states which include $s[b,j+1]$ are:

1. $\{(s[b,j+1],x) \mid x \in R_1\}$: these states are not feasible because message m was not yet sent but

* A state s_x is said to precede state s_y if there is a transition from s_x into s_y and s_y is not the initial state.

already received.

2. $\{(s[b,j+1],x) \mid x \in R2\}$: these states are not feasible because message m'' was not yet sent but already received.
3. state $(s[b,j+1],s[a,i+1])$ is feasible.

This example illustrates the rule for generating the set of all the global feasible states in a protocol where all the conversations are synchronized. Namely, for every non-external transition m add to the set of global feasible states the states $(s[a,i],s[b,j])$, $(s[a,i+1],s[b,j])$ and $(s[a,i+1],s[b,j+1])$. If any of the STDs contains transitions which are triggered by external events (e.g. an external request from an application program), then the following rule must be added. Let $Y(s)$ be the set of all the states which can be reached from state s through a path containing transitions associated with external events only. For every non-external transition m add to the set of global feasible states the states $(s[a,i],s[b,j])$, $(s[a,i+1],s[b,j])$, $(s[a,i+1],s[b,j+1])$ and the sets:

```
{ (s[b,j],q) | q ∈ Y(s[a,i+1]) },  
{ (s[b,j+1],q) | q ∈ Y(s[a,i+1]) } and  
{ (s[a,i+1],q) | q ∈ Y(s[b,j+1]) }.
```

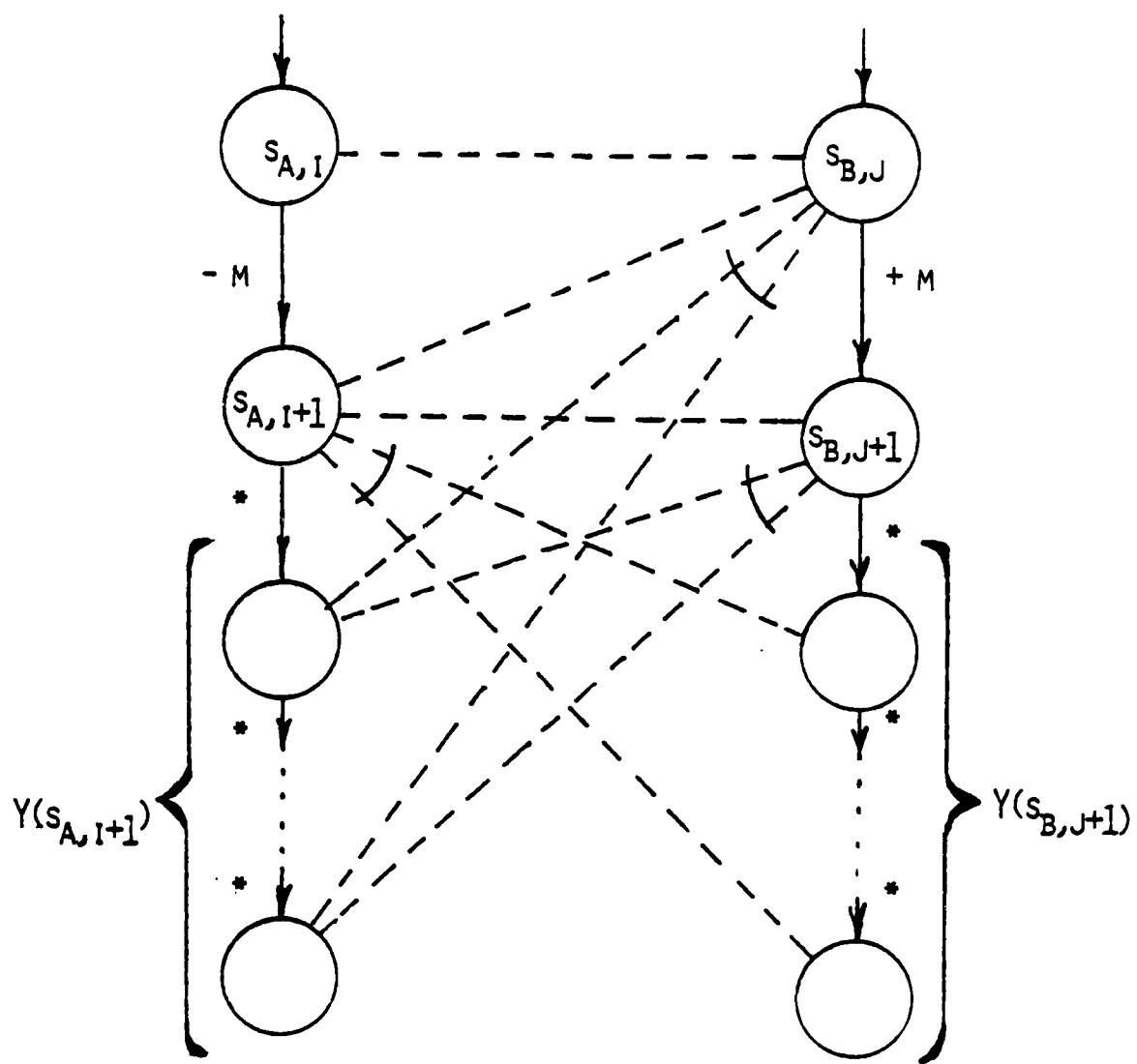


Figure 3.4 - Graphical representation of the set of rules to find the set of global feasible states associated with transition m . The dashed lines represent external events.

Figure 3.4 illustrates these rules. Using the rules derived above, one obtains the set of feasible states for the grant and release lock protocols in a system with an LC

| FEASIBLE GLOBAL STATES | (LC STATE; LLC _i STATE) |
|--------------------------------------|------------------------------------|
| (s ₂ , s ₁₁) | (0, 1, 0 ; 0, 0, 0) |
| (s ₃ , s ₁₂) | (0, 1, 0 ; 0, 1, 0) |
| (s ₂ , s ₁₆) | (0, 1, 0 ; 1, 0, 1) |
| (s ₄ , s ₁₃) | (1, 0, 0 ; 0, 1, 0) |
| (s ₅ , s ₁₄) | (1, 0, 0 ; 1, 0, 0) |
| (s ₇ , s ₁₃) | (1, 0, 1 ; 0, 1, 0) |
| (s ₇ , s ₁₄) | (1, 0, 1 ; 1, 0, 0) |
| (s ₈ , s ₁₅) | (1, 0, 1 ; 1, 0, 1) |
| (s ₉ , s ₁₆) | (0, 0, 0 ; 1, 0, 1) |
| (s ₁₀ , s ₁₁) | (0, 0, 0 ; 0, 0, 0) |

TABLE 1 - Feasible Global States

and one LLC. These states are shown in Table 1. In general, a global state for a system with an LC and k LLCs is of the form of $S = (X; x_1, x_2, \dots, x_k)$ where X is an LC-state and x_i , for $i=1, \dots, k$, is an LLC state compatible with the LC state X , according to table 1. For instance, if $X = s_7 = (1, 0, 1)$ the three following are examples of global states when we have three LLCs: $(s_7; s_{13}, s_{14}, s_{13})$, $(s_7; s_{13}, s_{13}, s_{14})$ and $(s_7; s_{13}, s_{13}, s_{13})$. The next section gives some definitions and proofs regarding the global feasible states of the LOCK tables, L-lists and R-lists.

3.2.2.4 - Some Definitions and Proofs

We will show here that, if no crash occurs, the Lock Granting and Lock Releasing algorithms have the property that a lock is only granted or released if all the sites in the component know about the request. In order to make this statement more precise consider the following definitions. Let $LT(i)$, $L(i)$ and $R(i)$ be the LOCK table, L-list and R-list at site i respectively.

DEFINITION 3.1 (Lock Request Presence): A lock request x or a lock is said to be present at site i if $x \in [LT(i) \cup L(i)] - R(i)$.

DEFINITION 3.2 (Release Request Presence): Let x be a lock request and let y be its associated lock release request. It is said that y is present at site i if $y \in R(i)$ or if $x \in LT(i) \cup L(i)$.

It is convenient at this point to define precisely the meaning of a site being relevant to a lock. We say that site i is relevant to lock x if at least one of the data items addressed or covered by x are stored at site i . Let us define $S(x)$ as the set of sites relevant to the lock x . We can now make the following statements.

ASSERTION 3.1: If a lock $x \in LT(LC)$ and if there is no pending release request associated with x , then the lock request x is present at every site in $S(x)$.

ASSERTION 3.2: If a lock $x \notin LT(LC)$ and if there is no pending lock request associated with x , then the release lock request y associated with x is present at every site in $S(x)$.

The proof for these two assertions, as well as for all other assertions in this chapter, can be found in Appendix A of this dissertation. Together, they lead directly to the following result.

THEOREM 3.1: Let C be a logical component, LC its lock controller and U the set of sites in C . If no crashes ever occur then a lock request is only granted by the LC after it is present at all the relevant sites in U and a lock is only released if the associated release request is present at every relevant site in U .

On a later section the results in this section will be extended to deal with situations in which crashes can occur.

3.2.3 - Crash Recovery

So far we have described the protocol for requesting locks and releasing them, assuming that no crash occurred. Communication links, processors, operating systems and processes are some examples of sources of crashes.

The three already mentioned recovery mechanisms will be presented here. These mechanisms will be proven to be

robust with respect to additional failures. To be robust, the protocols must preserve logical component internal and mutual consistency as defined below, if any changes have been made to any permanent information (like LOCK tables, up lists or LC id's) at any node.

DEFINITION 3.3 (LT-consistency): The set of LOCK tables of a Logical Component is said to be LT-consistent if assertions 3.1 and 3.2 hold at any time.

DEFINITION 3.4 (Logical Component Internal Consistency): A logical component is said to be internally consistent if the set of its LOCK tables is LT-consistent and if there is one and only one LC, whose identity is known to every node in the component.

DEFINITION 3.5 (Logical Component Mutual Consistency): A set of logical components is said to be mutually consistent if all of them are internally consistent and if there is no lock present at any LOCK table of one of them which conflicts with another such lock of any other component.

Definition 3.5 covers the previous two, and specifies an important property which is required of recovery.

The recovery protocols have been designed so that all crashes which can occur during a recovery phase fall into one of the two disjoint classes, which we call terminal and transparent failures.

A terminal crash causes the entire recovery mechanism to be aborted and restarted. The possible conditions under which terminal crashes occur are shown to leave the protocol in a robust state, as defined above. A transparent crash is defined to be one which does not affect the continued correct operation of the recovery process.

The following is the definition of robustness used in this chapter. A recovery protocol is robust if all crashes can be shown to be either terminal or transparent. As we will see, for each of the recovery mechanisms, we can identify a point before which the recovery can be considered as not having happened at all and after which it is considered to be successfully carried out. This point is called the completion point. Crashes before the completion point, if they have any effect at all, are shown to be terminal. Crashes after the completion point are shown to be transparent.

The three proposed recovery mechanisms will be shown to occur disjointly in time. In other words, a merge of two logical components only takes place if both are in their normal state or are not recovering from a Logical Component Crash. Also, a site only becomes attached to a logical component if this component is in its normal state. These important properties will allow us to state and prove separate theorems concerning each one of them.

3.2.3.1 - Logical Component Recovery (LCR)

We will now show how an LLC may become an LC if the LC crashes. A crash of the LC can be detected by any process engaged in a conversation or exchange of messages with it. As an example, an AP may time-out while waiting for a reply from the LC for a lock or lock release request. In every case, the process which detects a crashed LC is responsible for nominating a new LC. For this purpose, we will assume that the distinct sites or nodes in the underlying network are arranged in a linear order such that node #i precedes node #(i+1) mod n. Let this order be called the nomination order. So, whenever a process detects a failed LC it nominates the next node which is up in the nomination order after the crashed LC to the position of LC. This nomination is accomplished by the issue of an "ACCEPT NOMINATION" or AN message by the nominator. If this message is not acknowledged after a certain number of times it has been retransmitted, the nominator assumes that the nominee is down and sends an AN message to the next site in the nomination order. However, it may be the case that the originally nominated node was not down, as assumed by the nominator, but that due to certain conditions in the network its reply was seriously delayed. So, it seems that more than one LC could be nominated in this process! Let us neglect this issue for the moment, while we describe the recovery procedure, and show later how such an undesirable situation can be easily

avoided. The nominee is first responsible for checking that the old LC is actually dead (since the nomination may have come from an errant AP). Then the nominee must notify every other site that it has accepted the nomination. Moreover, the nominee must make sure that all the copies of the LOCK table be appropriately updated. From now on, we will refer to the crashed LC as the 'old LC' and to the nominee as the 'new LC'.

The process by which the new LC becomes the actual LC can be divided into two phases: a 'notification phase' and a 'LOCK table update phase'.

In the notification phase all the nodes in the component, as indicated by the up list U, are informed of the identity of the new LC. Also, in this phase enough information is gathered in order to appropriately update the LOCK tables in the subsequent phase. The update of the LOCK tables is done in such a way that the maximum forward progress is obtained at the end of the LCR procedure. More precisely, in terms of the STDs introduced in section 3.2.2.3, the LCR mechanism leaves the system in the global state which would have been reached by the system if a crash had not occurred and if no new requests were submitted.

The new LC, upon nomination, will issue a message called "NOMINATION ACCEPTED". This message will circulate once through the set of all sites in U (including the site

where the new LC runs) in a predetermined order. Notice that the up list U of the nominated LC determines the logical component to be restored by the Logical Component Recovery procedure.

During the NA cycle, two sets will be constructed, namely the set L of locks to be added to all the LOCK tables and the set R of locks to be deleted from all the LOCK tables. The set L includes all the locks which are in at least one lock pending list and that would therefore end up in all LOCK tables if no crash had occurred. The set R includes all the locks which are in at least one lock release list and would therefore be deleted from all LOCK tables in normal conditions.

It is possible for a given lock to be in the L-list at one site and at the R-list at another site. This situation can be seen from Table 1 and is also illustrated by the following scenario. Consider an LC and two LLC, LC1 and LC2. Consider a lock x which was accepted by LC1 and LC2 already. The LC then enters the lock into its LOCK table and sends the CONFIRM LOCK (CL) message to LC1 and to LC2. LC1 receives the message and moves the lock into its LOCK table. Now, the LC receives a release request for the same lock. It then sends an ACCEPT RELEASE (AR) message to LC1 and LC2. LC1 receives it and enters the request into its R-list. LC2 did not receive neither the CL nor the AR messages so far

and therefore the lock x is still in its L-list.

When constructing the sets L and R one has to be able to decide whether a lock should be installed in all the LOCK tables or deleted from all of them in those cases in which the request appears in both L-lists and R-lists. This decision is easily done with the aid of the sequence numbers which the LC attaches to every request. The greater the sequence number the later is the request. Therefore, the latest requests will be considered when constructing the sets L and R.

Every node in the NA cycle, other than the newLC, receives partially constructed sets L and R, adds its contributions to them and places the new versions of the sets into the NA message which is forwarded to the next node in the cycle. When the NA message returns to the newLC, the sets L and R are completed. The sets L and R are modified at site i according to the algorithm given below.

The actual update of the LOCK tables is done using the ACP protocol. Therefore if an additional crash occurs during the LOCK table update phase, the set of possible states in which the LOCK tables, L-lists and R-lists may be left is the same as the set of possible states which can result if the crash had occurred during normal operation. Therefore, recovery from additional failures merely requires res-

Algorithm to Build the Sets L and R

```
FOR x ∈ L(i) DO
    IF there is y ∈ R associated with x
    THEN IF sequence#(x) > sequence#(y)
        THEN BEGIN
            COMMENT lock request is the latest;
            L := L ∪ {x} ; R := R - {y} ;
        END;
    ELSE;
ELSE L := L ∪ {x};
FOR y ∈ R(i) DO
    IF there is x ∈ L associated with y
    THEN IF sequence#(y) > sequence#(x)
        THEN BEGIN
            COMMENT the release request is the latest;
            R := R ∪ {y} ; L := L - {x} ;
        END;
    ELSE ;
ELSE R := R ∪ {y};
```

tarting the LCR mechanism again.

After the notification phase is over, the new LC will send a message to every LLC asking them to update their LOCK tables. This message is called an "UPDATE TABLE" or UT message, and it carries within it the sets L and R. The actual set R contained in this message also includes lock release requests for transactions which are not local anymore. These transactions are first aborted and their locks released when the LOCK table is updated. Upon receipt of the UT message, the L-list of each site will be made equal to the set of locks in the set L which are relevant to the site. Analogously, the R-list of each site will be made equal to the set of locks in the set R which are relevant to the site. After this is done, each site sends a "READY TO UPDATE" message or RU message to the new LC.

After receiving a RU from every up site the new LC becomes the actual LC by notifying all the LLCs that they can resume their normal activity. For this purpose the LC broadcasts a "RESUME NORMAL ACTIVITY" or RNA message. The new value for U is the set of sites from which the LC received a RU message. This new value for U is included in the RNA message, thus allowing every node in U to know the composition of the set U. Also, upon receipt of the RNA message, the LOCK tables are updated accordingly to the L-lists and R-lists.

Let us now describe how we can guarantee that only one LC will emerge from the notification process. Recall that the nominator will nominate the first up node in the nomination sequence. Let us make the following definition:

DEFINITION 3.6 (trial sequence, T[j,k]): A trial sequence, T[j,k], is the sequence $i[1], i[2], \dots, i[k-1]$ of site numbers for which an "ACCEPT NOMINATION" message has been unsuccessfully sent by a nominator j, before j sent an AN message to site #k.

For every AN message sent from site #j to site #k we include the sequence T[j,k] as part of it. This sequence will also be included as part of the "NOMINATION ACCEPTED" message which circulates through the set of sites. The purpose of this is to allow any site to resolve any conflict that can arise due to the race conditions discussed earlier

in the chapter. Namely, it is possible that more than one LC was nominated and consequently more than one NA message (from distinct sources) would be circulating. Conflicts are resolved by giving preference to the last LC to be nominated. NA messages originated by other nominated LCs are killed when they are detected to belong to the improper LC. Therefore, each time an NA message is received by node i the algorithm, shown below in Algol-like notation, is executed.

Algorithm to process an NA message at site i.

Assume that in each site there is a trial list T, which is empty if no NA message has already been received in the present recovery phase and that will be updated by the algorithm below (as executed at site i). Let the variable NEWLC(i) be the identification of the new LC-site as imagined by node i. Let T[j,k] be the trial sequence included in the NA message.

```
IF T is empty or k < T
THEN BEGIN
    COMMENT either no NA message has already been received
          for this recovery process or the current NA
          message belongs to a more recently nominated LC;
    T := T[j,k] ; NEWLC(i) := k ;
    END;
ELSE Send "KILL NA" message to site k informing it that its
      cycle was aborted;
```

In many instances in this protocol we require a certain message to circulate through a set of nodes, as it is the case of the NA message. Let us call such messages 'circu-

lar messages'. They always have a source or generator who is responsible for sending it through a cycle. The underlying network protocols assure us that messages will not get lost while going from one site to another by the use of time-out and retransmission schemes. However, a circular message can still be lost if a node in the cycle crashes after receiving it but before being able to forward it. The loss of a circular message can be prevented by having each node in the cycle send to the circular message generator an acknowledgment for it, but only after it was forwarded to the next node in the sequence. Now, the source is able to detect a cycle interruption and it can appropriately resume it by sending the last copy of the message to the appropriate site*. This source acknowledgment scheme at the Centralized Lock Controller protocol level will be assumed to exist whenever a circular message is necessary.

It should be noted that if an application program issues a lock or release request and the LC fails before the request is present at every site, the request will never appear in the local LOCK table even after the LCR is completed. Therefore, APs should timeout for requests and resubmit them.

* This procedure can be optimized to a desired level by having only every i-th node send an acknowledgment back to the generator of the circular message. Then there is additional uncertainty over which node in the cycle failed. It is up to the generator to resolve the uncertainty.

The LCR mechanism can be described in an abstract form by the STDs in figure 3.5. Figure 3.5.a is the STD for the newLC and figure 3.5.b is the STD for any local lock controller participating in LCR. Let LC be the identification of the oldLC and let LT be its LOCK table. Let LC* be the newLC and LT* be its LOCK table. Associated with each state in the STDs of figure 3.5 is a 3-tuple of the form (a,b,c) where a, b and c are the LC identification, the newLC identification and the LOCK table, respectively.

3.2.3.2 - Proofs About LCR

We would like to prove now that the notification phase ends with one and only one LC having been successfully nominated, and that all sites know the correct new LC identification. As a first step we state assertions 3.3 and 3.4 which are concerned with the behavior of LCR given that no additional crashes occur.

ASSERTION 3.3: Given that no additional crashes occur during LCR, there will be one and only one LC whose identification is known to all sites in the component at the end of the notification phase.

Next, let a globally accepted lock (release) request be one which is in all L-lists (R-lists) of a logical component.

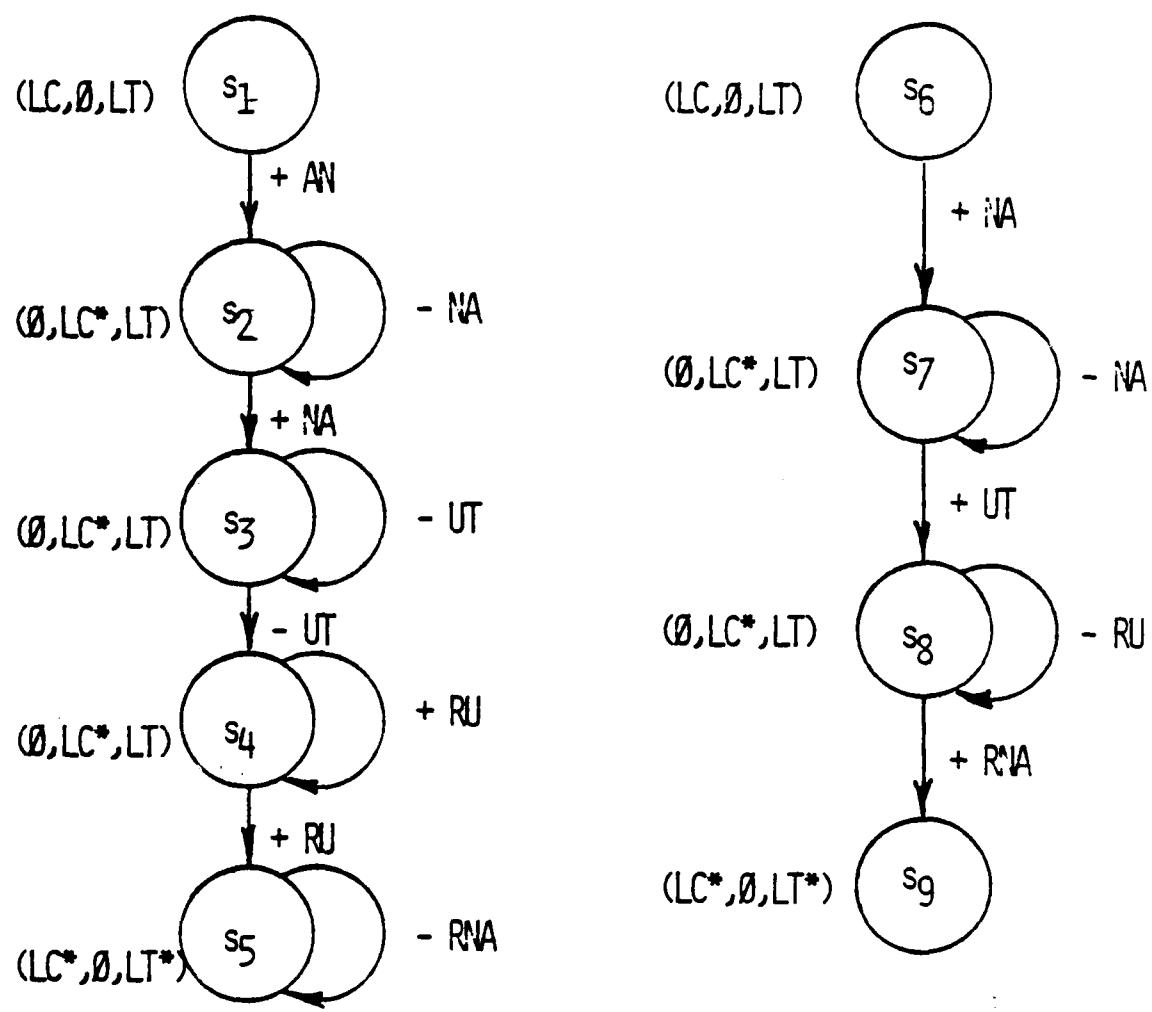


Figure 3.5 - STDs for LCR.

ASSERTION 3.4: (maximum forward progress - no additional crashes): Given that no additional crash occurs, the following is true at the end of the LCR mechanism. A lock p is in the LOCK table of all sites of $S(p)$ if the lock would have been in the LOCK table of the crashed LC if the crash had not occurred and if no new requests were submitted. Otherwise the lock is in none of the LOCK tables at the sites of $S(p)$.

Given these assertions we prove the robustness of the LCR mechanism.

THEOREM 3.2: The Logical Component Recovery (LCR) algorithm is robust.

Proof: The completion point for this algorithm occurs when the newLC has logged the fact that it is the LC after at least one RNA message has been sent. This point is indicated as state $s5$ in the STD of figure 3.5.a. The only terminal crash is a newLC failure before this point. This crash when detected will cause another LC to be nominated and the LCR mechanism to be restarted. In order to prove that LCR is robust we must prove that internal consistency is not violated by the partial execution of LCR. In other words we need to show that the set of LOCK tables of the component is LT-consistent. The crash of the newLC can occur:

- i) before any LOCK table has been updated.

ii) after some but not all LOCK tables have been updated.

iii) after all LOCK tables have been updated.

In case i) it is clear that the partially executed LCR has no effect at all. In case iii) all LOCK tables will be identical, therefore internal consistency for the component in question is trivially satisfied. In case ii) the crash occurred in the middle of the LOCK table update phase. But since LOCK tables are updated using the ACP protocol, the states of the LOCK tables, L-lists and R-lists at the instant of the crash is a state which could have resulted if the crash had occurred during normal operation. Therefore, the partial execution of the LCR mechanism does not violate the internal consistency of the component. The LCR mechanism will be restarted again as many times as crashes occur.

Now, it remains for us to analyze the transparent failures. Those are all the failures other than the newLC crash already discussed. We can have either a process or processor failure which simply knocks out one of the sites in the component, or the component can be partitioned into two or more components. In either case, a set of one or more nodes are isolated from the set of nodes which participate in the LCR mechanism. The nodes in this set will not be considered any more for the rest

of the LCR algorithm. However, we have to show that no inconsistencies are generated by a node dropping out during the execution of LCR.

For this purpose, we will examine all the possible instants at which a node j may crash.

CASE 1: during the 'nomination phase'

Here we have to show that the sets L and R will not be perturbed by any contributions already made to them by node j. Node j can crash at three possible instants.

CASE 1.1: before the NA message first reaches it.

In this case node j is simply removed from the up list U without contributing to the formation of either L or R.

CASE 1.2: after the NA message reaches it and before it is forwarded to the next node in the sequence.

Here, the node which sent the NA message to node j will timeout, detect its crash and send the NA message to the node which follows node j in the sequence. Again no contributions have been made to the sets L or R.

CASE 1.3: after the NA message has been forwarded

A crash of node j at this point is equivalent to a crash of a node during the 'LOCK table update' phase since node j already played its role in the 'notification

phase'. Therefore, this case reduces to the next one to be examined. The reader should notice that the robustness of this recovery mechanism relies heavily on the fact that the ACP protocol is used in the LOCK table update phase.

CASE 2: during the 'LOCK table update phase'

A crash of a node during this phase will have no effect upon other nodes, resulting only in the removal of this node from the up list of the logical component which is recovering

Examination of all these cases completes this proof. []

The above result allows us to relax the assumption made in assertion 3.4 that no additional crashes occur during LCR and state the following assertion.

ASSERTION 3.5: (maximum forward progress - additional failures allowed): Let C be a logical component. Let U be the up-list which defines the component C. Let U' be a subset of U which indicates the nodes which are actually up at the end of the LCR mechanism. The following is true at the end of the LCR mechanism. A lock p is in the LOCK table of all sites of $S(p) \cap U'$ if the lock would have been in the

* Note that $U \neq U'$ if any node crashes before the end of LCR but after all the TU messages have been received.

LOCK table of the crashed LC if the crash had not occurred and if no new requests were submitted. Otherwise, the lock is in none of the LOCK tables of the sites of $S(p) \cup U'$.

Finally we prove that every logical component is internally consistent.

THEOREM 3.3: Every logical component is internally consistent

Proof: Let C be any logical component. We have to prove that:

- i) the set of LOCK tables of C is LT-consistent
- ii) there is one and only one LC for C .

Statement i) is clearly true for normal operation of component C since assertions 3.1 and 3.2 were demonstrated for this case. Now, by assertion 3.5 either a lock p is in all the LOCK tables of $S(p)$ or it is in none of them at the end of LCR. Therefore, assertions 3.1 and 3.2 are trivially satisfied and consequently LT-consistency is preserved.

Statement ii) was proved to be correct in assertion 3.3 for the case in which no additional crashes occur during LCR. But, by theorem 3.2, LCR is robust. This allows us to consider the effect of LCR as if no additional crashes occur during its execution, and concludes

the proof [].

3.2.3.3 - Single Node Recovery

So far we have described how the system recovers from a logical component crash. We show now how a node which is down becomes active again, or in other words, how it gets logically connected to a logical component. Let node j be such a node. The first step to become active is to find out the identity of any LC. This step is carried out by sending the "WHO IS THE LC ?" or WLC message to any up node. Assume first that node j received at least one reply to its WLC message containing the identity of an LC. In this case, node j sends a message called "HI THERE" or HT to the LC telling him that node j is alive again. If the LC is not undergoing any kind of crash recovery it will send to node j the portion of its LOCK table relevant to node j as well as its up list. An "ACCEPT LOCK" or "ACCEPT RELEASE" message is sent to node j by the LC for every lock or release lock request for which not all the LA or RA messages have been received.

Assume now that node j does not get any answer or that all the nodes which replied to its WLC message are themselves recovering from a crash or coming up from normal system shutdown. Then, node j becomes a logical component on its own. Node j is the LC for this logical component and

the LOCK table, L-list and R-list are initialized as empty. The Logical Component Merge procedure described in section 3.2.3.5 will take care of integrating node j into another logical component.

3.2.3.4 - Robustness of SNR

THEOREM 3.4: The Single Node Recovery (SNR) algorithm is robust.

Proof: The only case of interest is the one in which the recovering node is able to get the identity of an LC as a response to its WLC message since, otherwise the SNR mechanism degenerates into a LCM as already explained above. Let j be the recovering node and let LC_i be the LC to which node j is trying to connect with. The proof is extremely simple since the only two crashes of interest are: a) LLC_j crash and b) LC_i crash. Case a) is clearly a terminal case. Case b) is also a terminal crash since a crash of LC_i, before it is able to send the LOCK table to LLC_j, prevents the LOCK table from being received by node j, thereby implying in SNR having to be restarted. This completes the proof. []

3.2.3.5 - Logical Component Merge

As a result of the Logical Component Recovery algorithm an LC will be elected in each logical component of the network. Transactions which are local to a component will continue to be serviced as if no disconnecting crash had occurred. On the other hand, transactions which span more than one component will have to wait until the components involved are brought together again. It is the responsibility of each LC to detect when two components are physically connected again and to take the necessary steps to merge them into one logical component. The merge of logical components will always be done on a pairwise basis. The whole Logical Component Merge mechanism is divided into two phases, namely a 'reconnection detection' phase and a 'merge' phase.

In the 'reconnection detection' phase, each LC sends periodically a "WERE YOU ALIVE" or WYA message to every node not in its up list. The purpose of this message is to detect the existence of sites which were not reachable before but which were up. For the purposes of the description that follows, let the two logical components to be merged be called C1 and C2. Let LC1 and LC2 be their respective LCs and U1 and U2 their respective uplists. LC1 will take an active role during the whole recovery phase, while LC2 will take a passive one. As we will see, a crash of LC1 while

the recovery mechanism is in progress will result in abort, while a crash of LC2 after the 'reconnection detection' phase is tolerated. Assume now that site #j in C2 received a WYA message from LC1. A component is said to be in NORMAL status if it is not undergoing any kind of crash recovery mechanism. If component C2 is in its NORMAL status, site #j sends a "YES I WAS" or YIW message to LC1. This message carries within it the identification of LC2.

At this point LC1 has to establish a logical connection with LC2. This connection is called a primary-secondary or P-S connection type with LC1 being the primary and LC2 the secondary. Since we require that LCM be done in a pairwise basis, the following conditions must be enforced by the protocol that establishes a P-S connection:

C1: an LC cannot be primary (secondary) for more than one P-S connection.

C2: an LC cannot be primary and secondary simultaneously.

The P-S connection is attempted by having LC1 send a "LET US MERGE" or LUM message to LC2. The status of LC1 is now changed to ATTEMPT. If the status of LC2 is NORMAL, which means that neither Logical Component Merge nor Logical Component Recovery is being attempted, LC2 sends a "MERGE ACCEPTED" or MA message to LC1 and changes its internal state to SECONDARY. Upon receipt of the MA message the con-

nection is considered to be successfully established by LC1. If the status of LC2 is not NORMAL then a "MERGE ATTEMPT REJECTED" or MAR message is sent to LC1 which will either retry later or will try a connection with another LC.

The above interconnection strategy could clearly allow undesirable race conditions to occur, such as having two LCs trying to play the role of primary, leading the system into deadlock situations. To avoid this problem, we assign a site dependent priority to each LC (no two sites have the same priority). LUM messages from lower priority LCs are rejected. LUM messages from higher priority LCs, if received while the connection has not yet been completed, i.e. the MA message has not been received, cause the connection being attempted to be broken. To this end the primary sends a "CLOSE CONNECTION" or CC message to its intended secondary.

That the protocol outlined above satisfies conditions C1 and C2 is proved in section 3.2.4.1. Figure 3.6 shows a state transition diagram describing the interconnection protocol. This protocol is the same for every node. Node labels are STATUSes, while arc labels are of the form R/T where R is the message whose arrival triggers the transition and T is a sequence of actions (transmission of messages) which occur as a consequence of the transition.

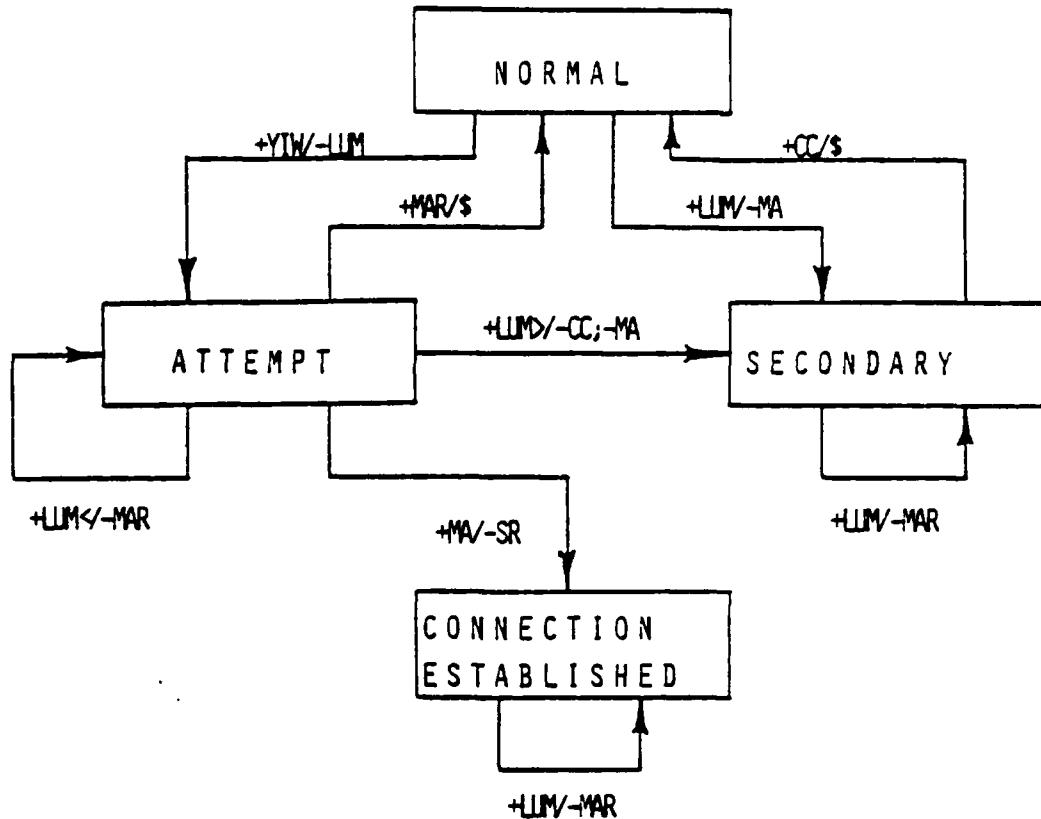


Figure 3.6 - STATE TRANSITION DIAGRAM FOR P-S CONNECTION ESTABLISHMENT. A plus (+) sign indicates reception of a message and a minus (-) sign indicates transmission of a message. The sign < indicates that the message in question originates from a lower priority source, while > indicates a higher priority site. The dollar (\$) sign indicates that no action is taken due to a state transition.

AD-A186 683

SECURE DISTRIBUTED PROCESSING SYSTEMS(U) CALIFORNIA
UNIV LOS ANGELES SCHOOL OF ENGINEERING AND APPLIED
SCIENCE G J POPEK DEC 78 UCLA-ENG-7955

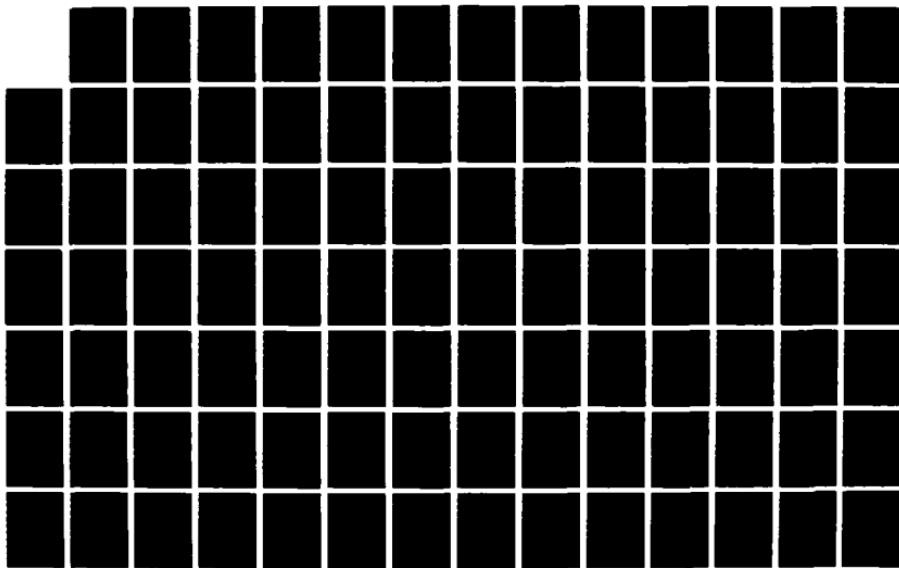
2/4

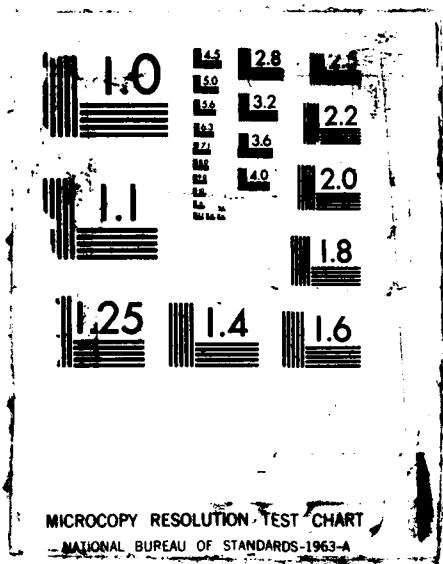
UNCLASSIFIED

MDA903-77-C-8211

F/G 12/7

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

After a P-S connection has been established between LC1 and LC2, they will not accept any more new lock or lock release requests from nodes in their components and will complete all outstanding ones. An outstanding request is one for which all AL or AR messages have been already sent but not all the corresponding LA or RA messages have been received. After all outstanding requests have been completed by LC2 it sends to LC1 a "READY TO MERGE" or RTM message containing as arguments the uplist U2 and the LOCK table at LC2 which now is the same for all nodes in C2. The receipt of the RTM message by LC1 marks the end of the 'reconnection detection' phase.

The 'merge' phase will construct the union of the LOCK tables at both components. Notice that up to this point no permanent change has been done to any LOCK table, nor uplist of any node. LC1 sends a "SUBSTITUTE YOUR TABLE" or SYT message for a cycle through the set of nodes in TEMP_U = U1 U U2. The SYT message is the agent which confirms the merge of the two components by taking within it the new LOCK table for the component. Also, the up lists are updated and LC1 becomes the new LC of the new logical component.

3.2.3.6 - Robustness of LCM

THEOREM 3.5: The Logical Component Merge (LCM) Algorithm is robust.

Proof: The completion point for the LCM algorithm is the point where the SYT message has already been received and accepted by one LLC.

Let $LT(i)$, $U(i)$ and $LC(i)$ be respectively the LOCK table at site i , the up list at site i and the LC identification as known by site i . It is worth observing that changes to the values of $LT(i)$, $U(i)$ and $LC(i)$ at any site i other than the LC-1 site are only done upon receipt of the SYT message.

Let us examine the possible cases of crashes before the completion point:

CASE 1: crashes during the 'reconnection detection' phase

A crash of either LC1 or LC2 in this phase will cause LCM to be aborted and a LCR to be started at the component who had an LC-crash. Since no LOCK table nor up list has been changed so far, this is a terminal crash. Since LC1 and LC2 are the only processes involved in this phase, we conclude that this phase is robust.

CASE 2: crashes during the 'merge' phase

A crash of LC1 during this phase will interrupt LCM and start LCR for component C1. As no permanent changes have been done already, this is a terminal crash. A crash of any other node (including LC2) clearly does not affect any other node nor the mutual consistency of the merged logical component [].

3.2.4. - Disjointness of the Recovery Algorithms

We show here that there is no interaction between the three recovery algorithms. To that effect one has to show that:

- a) LCM is done pairwise
- b) LCR, LCM and SNR are mutually exclusive.

To verify condition a) we only need to show that conditions C1 and C2 stated in section 3.2.3.5 are satisfied by the P-S connection protocol. This verification is done in section 3.2.4.1. Condition b) is shown to hold in section 3.2.4.2.

3.2.4.1 - Disjointness of LCMs

Consider a directed graph G whose vertex-set is the set of LCs and which has two distinct types of arcs, namely e-arcs and a-arcs. There is an e-arc from vertex i to vertex j if there is an established P-S connection between vertices

i and j, vertex i being the primary. Equivalently, an e-arc from vertex i to vertex j is said to be created in G whenever vertex i enters the CONNECTION ESTABLISHED state (see figure 3.6). There is an a-arc from vertex i to vertex j if vertex i is attempting a P-S connection to vertex j. Such an a-arc is created as soon as vertex i enters the ATTEMPT state (see figure 3.6). The graph G displays the pattern of established and attempted connections. Let e-G be the subgraph obtained from G by considering only e-arcs of G and a-G be the one obtained by taking only the a-arcs.

Conditions C1 and C2 can now be rephrased as follows:

C1.1: $0 \leq \text{indegree}(v) \leq 1$ and $0 \leq \text{outdegree}(v) \leq 1$ for all v in e-G.

C2.1: $\text{indegree}(v) * \text{outdegree}(v) = 0$ for all v in e-G.

Every a-arc will either be deleted from G when the attempted connection is broken or will become an e-arc if the connection is successfully established. So, we want to prove the following:

THEOREM 3.6: Given a graph G whose e-graph satisfies conditions C1.1 and C2.1, the new e-graph obtained from G as new connections are established also satisfies those conditions.

Proof: It can easily be seen, from the protocol specification, that condition C1.1 is satisfied not only by the initial e-graph but also by the graph G, since:

- a) if there is already a connection between vertices i and j or one is being attempted, no new connection is attempted by neither vertex i nor vertex j.
- b) if a connection has already been established or is being attempted, the secondary will reject all further attempts.

So, it remains for us to examine all the possible cases in which condition C2.1 could conceivably be violated in G and show that the resulting e-graph obtained when one or more a-arcs become e-arcs still satisfies this condition. There are four possible cases, two of which can never happen due to the protocol specification, while the remaining two have to be examined. Given any three vertices a, b and c, the four possible cases are:

- a) (a,b) and (b,c) are e-arcs.
- b) (a,b) is an e-arc and (b,c) is an a-arc.
- c) (a,b) is an a-arc and (b,c) is an e-arc.
- d) (a,b) and (b,c) are a-arcs.

Cases a) and b) are the impossible ones. In case c) the attempted connection between a and b will fail since there is an established connection from b to c (see the self

loop at the CONNECTION ESTABLISHED state of the diagram of figure 3.6). Therefore, arc (a,b) will disappear. In case d) nodes a and b are in the ATTEMPT state. If (a,b) becomes an e-arc we can see that the transition labeled LUM/CC;MA from state ATTEMPT to the state SECONDARY is taken at vertex b, causing the attempted connection (b,c) to be broken. Therefore arc (a,b) becomes an e-arc while arc (b,c) disappears. On the other hand, if (b,c) becomes an e-arc in the first place we are back to case c) which was already examined. []

We take the opportunity here to prove that the P-S connection protocol is such that all the a-arcs in G will, in a finite time, (of the order of magnitude of the transmission delay time in the network) either disappear or become e-arcs. In other words, the P-S connection protocol is deadlock free.

THEOREM 3.7: The P-S connection protocol is deadlock free.
Proof: We must prove that there can be no long lasting cycles in G. The interesting case is, of course, that of cycles made out only of a-arcs, since as shown in the previous theorem, any a-arc adjacent to an e-arc will disappear in a finite time.

Consider a cycle in a-G and two adjacent a-arcs (a,b) and (b,c) in the cycle. Vertices a and b are in the ATTEMPT state. There are only two possible cases to consider:

CASE 1: [PRIORITY(a) > PRIORITY(b)]: In this case, if the "MERGE ACCEPTED" message from vertex c is received by b before the "LET US MERGE" message from a then (b,c) becomes an e-arc and (a,b) disappears.

CASE 2: [PRIORITY(a) < PRIORITY(b)]: Here, arc (a,b) will disappear since a has lower priority than b.

In any event, the cycle will be eventually broken. Note also, that vertex c could be the same as a and the above analysis is still valid. []

3.2.4.2 - Disjointness of LCR, LCM and SNR

We first define a node state transition diagram as a directed graph whose vertices are states of a network node and whose arcs represent transitions between states. The state of a node i is the 3-tuple [STATUS(i), LC(i), U(i)], where LC(i), U(i) are as defined before. STATUS(i) is the status of the component to which site i is attached as viewed by site i. NORMAL status indicates that neither LCR nor LCM is in progress; RECOVERY means that LCR is taking place and QUIESCENT indicates that LC(i) is rejecting further requests. The labels on the arcs specify the conditions upon which a transition between two states occurs. These conditions can either be a crash detection or a message arrival. The diagram, shown in figure 3.7, shows all possible state transitions for a node, other than LC1, which is in a

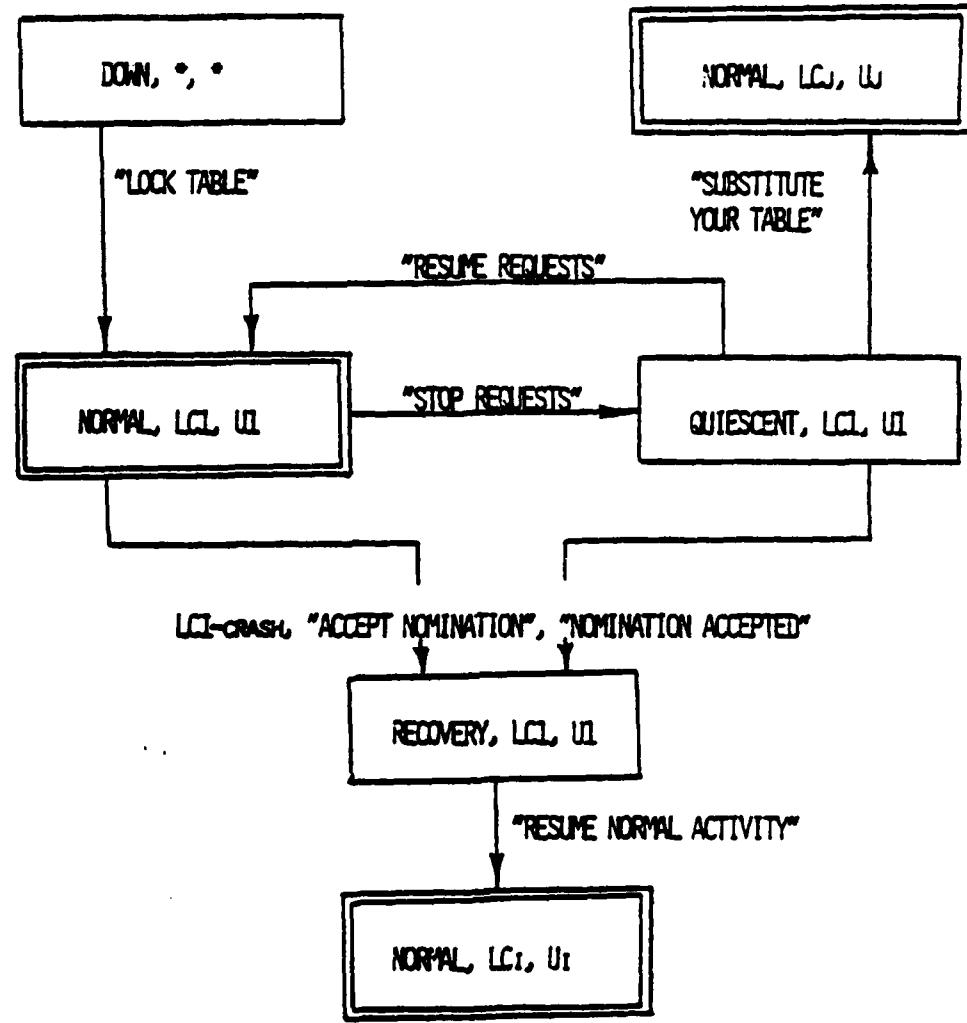


Figure 3.7 - NODE STATE TRANSITION DIAGRAM. The following relationships are observed:

- $U_j = U_1 \cup U_2$ where LC_2 is in U_2 .
- $U_1 \subsetneq U_1$.
- LC_1 is in U_1 .

component C1, with LC equal to LC1 and up list equal to U1. From every state there is a transition to the DOWN state. These transitions are not represented in the diagram for obvious reasons.

The state [NORMAL, LCj, Uj] is state which resulted from a successful merge of component C1 with another component, for instance C2. The state [NORMAL, LCi, Ui] is a state which resulted from a successful Logical Component Recovery.

By inspection of the diagram, we observe that a node can only go from one normal state to a different normal state after one and only one recovery mechanism has been completed. Therefore, there is no interaction among the three recovery mechanisms.

3.2.5. - Logical Component Mutual Consistency

Let us show here that the CLC protocol (including the recovery mechanisms) is such that the set of Logical Components into which the network is partitioned is mutually consistent.

THEOREM 3.8: The set of logical components into which the network is partitioned is mutually consistent.

Proof: By theorem 3.3 each one of the logical components

is internally consistent. It remains for us to prove that there can be no lock present at any LOCK table of any component which conflicts with another such lock of any other component. This theorem is trivially true when there is only one logical component. Further net partitioning does not destroy this property since locks are only granted if they are local to a component, which implies that they do not conflict with any other lock granted at any other component. []

3.2.6. - Database Consistency

We show here that given a deadlock free, consistency preserving locking mechanism for a centralized database (CDB), the CLC protocol can be used to implement an equivalent robust, deadlock free, consistency preserving locking mechanism for a distributed database (DDB). A database is said to be in a consistent state if all the data items satisfy a set of assertions or consistency constraints. A transaction is a sequence of accesses which take the database from a consistent state into another consistent state. Thus, a transaction is the unit of consistency. Let us define an access as the pair (P, a) where P is a logical description of the portion of the database to be accessed and a is an access mode (e.g. read, write, delete, etc.). If all the locks are granted by a process which has complete knowledge of every other active

locks (as is the case with the LC) and if every access is checked against the LC copy of the LOCK table (this condition will be relaxed later), to see whether the transaction holds the necessary locks , then the 'lock scheduler' for a CDB described by Eswaran [1] can be implemented in a straightforward manner with the use of the CLC protocol. Such a locking mechanism has the properties of being robust and preserving the consistency of the DB. Notice that deadlock prevention or detection mechanisms can be carried out by the LC since it has complete control over all activities in its component. Recall that if the network is partitioned into more than one component, locks granted in one of them do not conflict with locks active in others. Therefore, distinct LCs manage disjoint sets of "resources", where a resource here means an individually lockable data item in the DB. So, a deadlock prevention or detection policy can be implemented in each LC independently of all the others.

The requirement that every access be checked against the LOCK table at the LC-site can be relaxed in favor of having the access checking done locally. In order for this to be possible a lock must be considered to be active at a given site i for a time interval T2 contained in the time interval T1 during which the lock is active at the LC-site, otherwise some portions of the DB could be locked in conflicting modes for different transactions. Figure 3.8 shows

a double time axis diagram displaying time at the LC-site and at a given site i where a lock request is originated. T_1 starts when the "CONFIRM LOCK" message is sent to every site in the component and ends with the broadcast of the "CONFIRM RELEASE" message. T_2 starts with the arrival of a CL message at site i . Although a lock is only removed from a LOCK table when the corresponding "CONFIRM RELEASE" message arrives, it can be flagged as 'waiting for removal' as soon as a "RELEASE LOCK" message is sent from the LC to site i . For access checking purposes, all flagged locks must be considered as non active. The extra precaution that must be taken in this case is to unflag all flagged locks after LCR has taken place.

3.2.7 - Cost and Delay Analysis

An analysis of the cost and delay associated with the CLC protocol is presented in this section. The types of costs we can consider are:

- communication cost: this is the cost associated with the exchange of messages required by the protocol.
- processing cost
- storage overhead cost

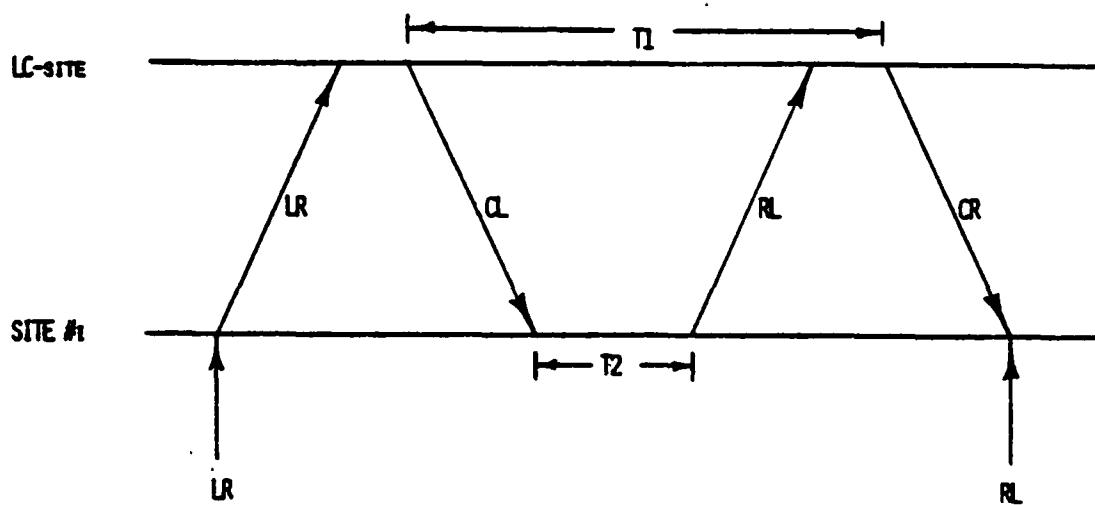


Figure 3.8 - T_1 is the time interval during which a lock is considered to be active at the site where the LC is located. T_2 is the time during which the same lock is considered to be active at the requesting site # 1.

3.2.7.1 - Delay Analysis

This section considers the delay, introduced by the CLC protocol, for a lock request to be granted, DL, and the delay, DR, for a lock to be released. The average delay, Dupdt, for an update request to be completed is also calculated for the update model described here.

Let,

TMAX = average maximum delay introduced by the network.

T = average message delay introduced by the network.

W = average waiting time for a lock request to be granted at the LC.

Dupdt = average delay for an update given that no crash occurs during the whole period.

DL = average delay for a lock request to be granted under no crash conditions.

DR = average delay for a lock to be released under no crash conditions.

n = number of sites.

m = number of sites participating in a transaction.

We will use the notation X → Y : M to denote the fact that process X sends message M to process Y. The processes

to be considered here are the set of APs (application programs), the LC (lock controller) and the set of LLCi's (local lock controllers).

Let us first calculate DL.

$$\begin{aligned} DL &= T + (AP \rightarrow LC : LR) \\ &\quad W + (\text{delay due to conflicts in LT}) \\ &\quad TMAX + (LC \rightarrow LLCi : AL) \\ &\quad TMAX + (LLCi \rightarrow LC : LA) \\ &\quad T (LC \rightarrow AP : LG) \\ &= 2*(T + TMAX) + W \end{aligned} \tag{3.1}$$

We have neglected in the above calculations the processing time at the several sites. Also, the exchange of messages between LC and every other LLCi takes place in parallel. This gives rise to the term TMAX in (3.1).

Let us now calculate DR.

$$\begin{aligned} DR &= T + (AP \rightarrow LC : RL) \\ &\quad TMAX + (LC \rightarrow LLCi : AR) \\ &\quad TMAX + (LLCi \rightarrow LC : RA) \\ &\quad T (LC \rightarrow AP : RG) \\ &= 2*(T + TMAX) \end{aligned} \tag{3.2}$$

In order to evaluate Dupdt we will use a model for update in which the lock and release requests and their corresponding messages travel together with the update re-

quest in a single physical message. The following calculation shows explicitly the exchange of messages involved here.

$$\begin{aligned} \text{Dupdt} &= T + (\text{AP} \rightarrow \text{LC} : \text{LR+update request+RL}) \\ &\quad W + \text{TMAX} + (\text{LC} \rightarrow \text{LLCi} : \text{AL+AR}) \\ &\quad \text{TMAX} + (\text{LLCi} \rightarrow \text{LC} : \text{LA+RA}) \\ &\quad \text{TMAX} + (\text{LC} \rightarrow \text{LLCi} : \text{CL+do update+CR}) \\ &\quad T + (\text{LC} \rightarrow \text{AP} : \text{update done}) \\ &= 2*T + 3*T\text{MAX} + W \end{aligned} \tag{3.3}$$

Note that although some previously defined messages are now grouped together into a single physical message, as indicated additively in the above calculations, they are processed as separate messages and as if they were received in the above specified order. For instance, CL+do update+CR is equivalent to three messages, namely CL, do update and CR, received in this order. The update is performed at the LC-site upon receipt of all the LA+RA messages and it is performed at each of the remaining sites upon receipt of the CL+do update+CR message.

Let us now calculate the average delay, R, involved in the Logical Component Recovery mechanism. As was discussed earlier, the crash recovery is divided into two phases: the 'notification phase' and the 'LOCK Table update phase'.

Let

R1 = average delay in the notification phase.

R2 = average delay in the LOCK Table update phase.

The nominator will send an "ACCEPT NOMINATION" message to the nominee. If it does not get a reply it will send another AN to the next site in the nomination order, after a time out, Tout. If we let k be the number of sites to which an AN message was sent until a reply is received, we can calculate R1 as,

$$\begin{aligned} R1 &= (k - 1) * \text{Tout} + \\ &\quad T \quad + (\text{nominator} \rightarrow \text{nominee : AN}) \\ &\quad n * T \quad (\text{AN message cycles through set of sites}) \\ &= (k - 1) * \text{Tout} + (n + 1) * T \end{aligned} \tag{3.4}$$

Now, R2 is

$$\begin{aligned} R2 &= \text{TMAX} + (\text{newLC} \rightarrow \text{LLCi : UT}) \\ &\quad \text{TMAX} + (\text{LLCi} \rightarrow \text{newLC : UT}) \\ &\quad \text{TMAX} \quad (\text{newLC} \rightarrow \text{LLCi : RNA}) \\ &= 3 * \text{TMAX} \end{aligned} \tag{3.5}$$

Finally, R is

$$\begin{aligned} R &= R1 + R2 = \\ &= (k-1) * \text{Tout} + T * (n + 1) + 3 * \text{TMAX} \end{aligned} \tag{3.6}$$

If we let $T_{out} = a*T$ where a is constant ($a > 2$) we have

$$R = [a*(k-1) + n + 1]*T + 3*T_{MAX} \quad (3.7).$$

Since $1 \leq k < n$ we can find the following lower and upper bounds for R .

$$R \geq (n + 1)*T + 3*T_{MAX}$$

and

$$R < [a*(n-2) + n + 1]*T + 3*T_{MAX} \quad (3.8).$$

3.2.1.2 - Cost Analysis for CLC Protocol

We will neglect the processing cost here. Let M be the average communication cost per message. Let us assume here that the site at which the LC resides is one of the sites participating in the transaction. If this is not the case, one must add one extra message per broadcast message in the protocol.

The number of messages required to have a lock granted under no crash conditions, N_l , is

$$\begin{aligned} N_l = 1 &+ (AP \rightarrow LC : LR) \\ &+ (m-1) + (LC \rightarrow LLC_1 : AL) \\ &+ (m-1) + (LLC_1 \rightarrow LC : LA) \\ &+ 1 + (LC \rightarrow AP : LG) \\ &+ (m-1) + (LC \rightarrow LLC_1 : CL) \end{aligned}$$

$$= 3^m - 1 \quad (3.9)$$

Also, the number Nr of messages necessary to release a lock is:

$$\begin{aligned} \text{Nr} = & 1 + (\text{AP} \rightarrow \text{LC} : \text{RL}) \\ & (m-1) + (\text{LC} \rightarrow \text{LLCi} : \text{AR}) \\ & (m-1) + (\text{LLCi} \rightarrow \text{LC} : \text{RA}) \\ & 1 + (\text{LC} \rightarrow \text{AP} : \text{RG}) \\ & (m-1) (\text{LC} \rightarrow \text{LLCi} : \text{CR}) \\ = & 3^m - 1 \end{aligned} \quad (3.10)$$

Using our previously defined update model, the number of messages exchanged in order to perform an update, Nupd_t, is

$$\begin{aligned} \text{Nupd}_t = & 1 + (\text{AP} \rightarrow \text{LC} : \text{LR+update request+RL}) \\ & m-1 + (\text{LC} \rightarrow \text{LLCi} : \text{AL+AR}) \\ & m-1 + (\text{LLCi} \rightarrow \text{LC} : \text{LA+RA}) \\ & m-1 + (\text{LC} \rightarrow \text{LLCi} : \text{CL+do update+CR}) \\ & 1 (\text{LC} \rightarrow \text{AP} : \text{update done}) \\ = & 3^m - 1 \end{aligned} \quad (3.11.1)$$

Therefore the average communication cost per update, Cupd_t is,

$$\text{Cupd}_t = (3^m - 1) * M \quad (3.11.2)$$

where again m is the number of sites participating in the transaction and not the total number of sites in the net-

work.

Let us now calculate the recovery cost, C_{rec} . The number of messages, N_1 , exchanged during the recovery phase given that k AN messages were sent by the nominator can be calculated as follows.

$$\begin{aligned} N_1 &= k + (k \text{ AN messages}) \\ &\quad n + (\text{cycle of NA message}) \\ &\quad (n-2) + (\text{source ACK messages}) \\ &\quad (n-1) + (\text{newLC} \rightarrow \text{LLC}_1 : \text{UT}) \\ &\quad (n-1) + (\text{LLC}_1 \rightarrow \text{newLC} : \text{UT}) \\ &\quad (n-1) (\text{newLC} \rightarrow \text{LLC}_1 : \text{RNA}) \\ &= k + 5^*n - 5 \end{aligned} \tag{3.12}$$

As $1 \leq k < n$, we can find the following lower and upper bounds for C_{rec} .

$$\begin{aligned} C_{rec} &\geq (5^*n - 4)^*M \\ \text{and} \\ C_{rec} &< (6^*n - 6)^*M \end{aligned} \tag{3.13}$$

3.2.8. - Extension

It has been observed in most of the existing distributed systems that a large percentage of the generated transactions is local, in the sense that the resources needed to satisfy a given transaction are either located at the site of origin of the transaction or in neighboring sites. This

observation suggests that significant savings in terms of communications cost and delay can be achieved if one optimizes the operation of the algorithm to adapt to such a highly skewed distribution of activity. To illustrate the point, consider a set of interconnected computer networks. We believe that in such a case, most of the operations will be confined to one computer network while relatively few operations will cross network boundaries.

This section outlines an extension to the CLC protocol that permits the forms of performance optimization needed for the cases discussed above. The extension, which we call an HCLC (for Hierarchical CLC) protocol, consists of a hierarchical organization of resource controllers. A tree of controllers is provided where the root is considered to be at level 0 and all the children of a controller at level i are at level $i+1$ in the hierarchy.

Each controller (except for the leaves) serves as an LC for its children. Also, each controller (except for the root of the hierarchy) acts as an LLC for its parent. Therefore, each controller has to maintain two distinct LOCK tables, which we call parent-LT and child-LT. The parent-LT for the root controller contains one lock for the whole DB in exclusive mode. The child-LT for a leaf is empty.

An intuitive description of the normal operation of the HCLC protocol can be easily understood in the light of an

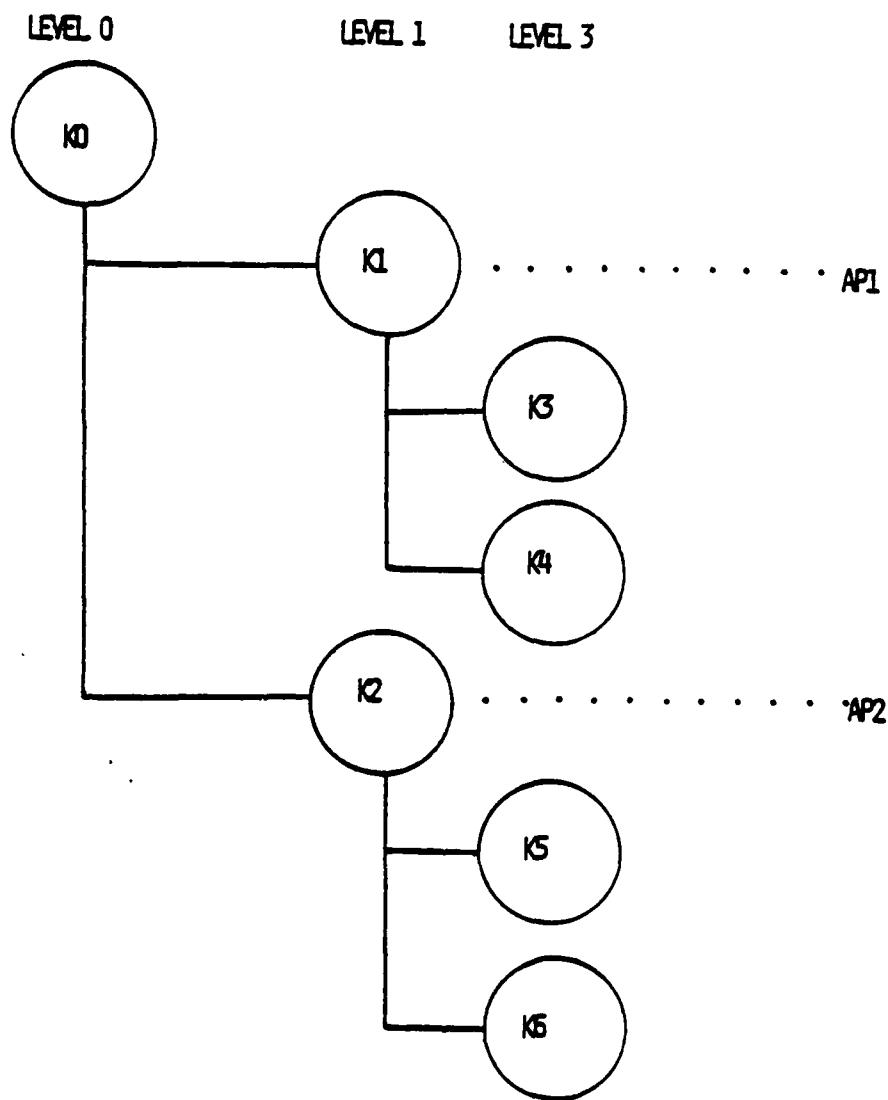


Figure 3.9 - HIERARCHY OF LOCK CONTROLLERS. AP1
and AP2 are application programs.

example. Figure 3.9 shows a three-level hierarchy. Application programs interact with lock controllers K1 and K2 at one level above the leaves (since the leaves are LLCs). This interaction is the same as the AP-LC interaction in the CLC protocol. Actually, application programs are not aware of the fact that the controllers are hierarchically organized. Let a lock request, x , from AP1 be submitted to K1. If x conflicts with any other lock in child-LT(K1) then the lock request is treated in the same way as in the CLC protocol. If there is no conflict, K1's parent-LT is searched for a lock y which covers x . A lock x_1 is said to cover a lock x_2 if the portion of the DB specified by x_2 is contained in the portion of the DB addressed by x_1 and if the lock mode specified by x_1 is not weaker than the lock mode in x_2 . The existence of a lock such as y in parent-LT(K1) indicates that K1 currently has control over the resources requested by AP1. If y is found, the lock request x can be granted and to this end K1 interacts with K3 and K4 in the same way as an LC interacts with the LLCs in its component. On the other hand, if y cannot be found, the lock request x is submitted by K1 to K0. K0 will act with respect to K1 and K2 in the same way that K1 did with respect to K3 and K4. The difference in this case is that since K0 is the root there is a lock in parent-LT(K0) for the whole DB in exclusive mode. This lock covers any other lock.

In an HCLC protocol, locks may be released either explicitly or automatically. Locks in child-LT(K_i), for $i=1,2$, are released explicitly upon request from APs using the same mechanism described in the CLC protocol. Locks in parent-LT(K_i), for $i=1,2$, can be released automatically as soon as there are no locks in the corresponding child-LTs which depend upon them. To this end, each lock y , in parent-LT(K), for any controller K , has associated with it a list of locks in child-LT(K) covered by y . Also, each lock x in a child-LT(K) points to the lock y in parent-LT(K) which covers x . When a lock x is explicitly released from child-LT(K_1) the lock list for its corresponding lock, y , in parent-LT(K_1) is appropriately updated. Whenever this list becomes empty, a release request may be automatically generated by K_1 and submitted to K_0 . In general, the automatic release of locks can be propagated up to the root.

This hierarchical protocol can be easily adjusted by policy decisions both to delay such releases, and to establish early locks at higher levels in anticipation of local lock requests. Lock management analogous to LRU-like memory management policies are obvious policy candidates.

For the set of interconnected computer networks, a three-level hierarchy could be constructed as follows. There is one LC per computer network, all of them at level 1. Their children, at level 2, are their corresponding

LLCs. Finally, the root is any site acting as a global controller for the entire collection of computer networks.

An interesting property of the proposed extension is that there is always one controller which is able to detect the existence of a cycle in the lock-request graph. This controller is the common ancestor, with the largest level number, to all the controllers where requests in the cycle were originated. In the example of figure 3.9, the common ancestor to K1 and K2 is K0.

Crash recovery algorithms for the HCLC protocol must include mechanisms to reconstruct the hierarchy, in addition to the recovery mechanisms present in the CLC protocol.

3.3 - Deadlocks in Distributed Databases

This section is concerned with issues of locking and deadlock detection mechanisms in distributed databases. The problem of system deadlocks in multiprogramming and multiprocessing systems has received considerable attention in the literature and is well understood [HABE 69, SHOS 70, COFF 71]. There are three approaches to the treatment of deadlocks: deadlock prevention, deadlock avoidance and deadlock detection and resolution.

Deadlock prevention requires that all the resources be acquired at once by a transaction. This requirement cannot

always be satisfied in a database environment since the resource needs of a transaction may be data dependent and not precisely known at the start of the transaction. Therefore it would be necessary for a transaction to acquire all possible resources required thereby decreasing system concurrency.

Deadlock avoidance requires some advance knowledge of the resource usage of transactions in order to determine at each point in time whether there is a valid sequence of actions of the already initiated but not yet completed transactions such that all of them can be run to completion. Again, this approach is not practical in distributed databases since the necessary advance information to avoid deadlocks is either absent or is distributed enough to render inefficient any attempt to avoid deadlocks.

Deadlock detection can be done by searching for cycles in a "state graph" [COFF 71]. A method for the detection and resolution of deadlocks in a centralized database system was presented in [KING 73]. In distributed databases, however, it is not efficient to maintain a global state graph for the whole system. Two methods for detecting deadlocks in distributed databases, which do not require that a global graph be built and maintained, are presented in this section - one hierarchical and one distributed. An outline of the proof that both protocols will detect all existing deadlocks

is given here. For the case of the hierarchical protocol the problem of establishing the hierarchy in a way such that minimizes the cost of using the protocol is introduced.

3.3.1 - Formal Model of Transaction Processing

This section introduces the necessary notation and formalism upon which we base the locking protocols and deadlock detection mechanisms presented in this chapter. The database is considered to be distributed among n sites, S₁, S₂, ..., S_n, of a computer network. Users interact with the database via transactions. A transaction is a sequence of actions which can be either read, write, lock or unlock operations. Transactions are assumed to be two-phase, i.e. once an unlock operation was issued no other lock can be requested by the transaction. As shown in [ESWA 76] this is necessary to preserve the database consistency. If the actions of a transaction involve data at a single site the transaction is called local as opposed to a distributed transaction which involves resources at several sites. We assume that distributed transactions are implemented as a collection of processes which act on behalf of the transaction. Those processes are called transaction incarnations. There may be one or more incarnations of the same transaction at each participating site. A transaction incarnation is responsible among other things for:

- a. acquiring, using and releasing resources local to

the site at which it is executing as needed by the transaction.

- b. exchanging messages with remote incarnations of the same transaction for purposes of cooperating with incarnations located at foreign sites.

Note that this model of a transaction execution is general enough to accommodate other models.

A transaction can be in two different states, namely active and blocked. A transaction is blocked if its execution cannot proceed because a needed resource is being held by another transaction, and the transaction is active otherwise. We introduce now a graphic model which depicts the state of execution of all transactions in the system. This model is in the form of a graph called the transaction wait for graph or TWF graph. The nodes of this graph are associated with transaction incarnations and are labeled by the pair (transaction_name, site_name). Note that labeling transaction incarnations with the pair (transaction name, site name) provides unique global names for these nodes if there is at most one transaction incarnation per site per transaction. If more than one incarnation is to be allowed then distinct local names should be assigned for them. Since this distinction is irrelevant for the forthcoming discussion we will assume that there is only

one incarnation of any transaction per site.

- there is a directed arc from node (Ti, S) to node (Tj, S) if the incarnation of transaction Ti at site S is blocked and waiting for the incarnation of transaction Tj to release a resource needed by Ti . (Ti, S) is said to be in "resource wait" for (Tj, S) .
- there is a directed arc from node (T, Si) to node (T, Sj) if the incarnation of transaction T at site Si is blocked and waiting for a message from the incarnation of T at site Sj . (T, Si) is said to be in "message wait" for (T, Sj) .

It can be easily seen that the existence of a cycle in the `transaction_wait_for` graph is a necessary and sufficient condition for a deadlock to occur.

Figure 3.10 shows an example of a `transaction_wait_for` graph for a network with two sites S_1 and S_2 . This graph shows two deadlock cycles. One of them is a local deadlock since it involves only incarnations of transactions at site S_2 and therefore only resources local to S_2 . The other deadlock cycle spans both sites and is an example of what we call a global deadlock.

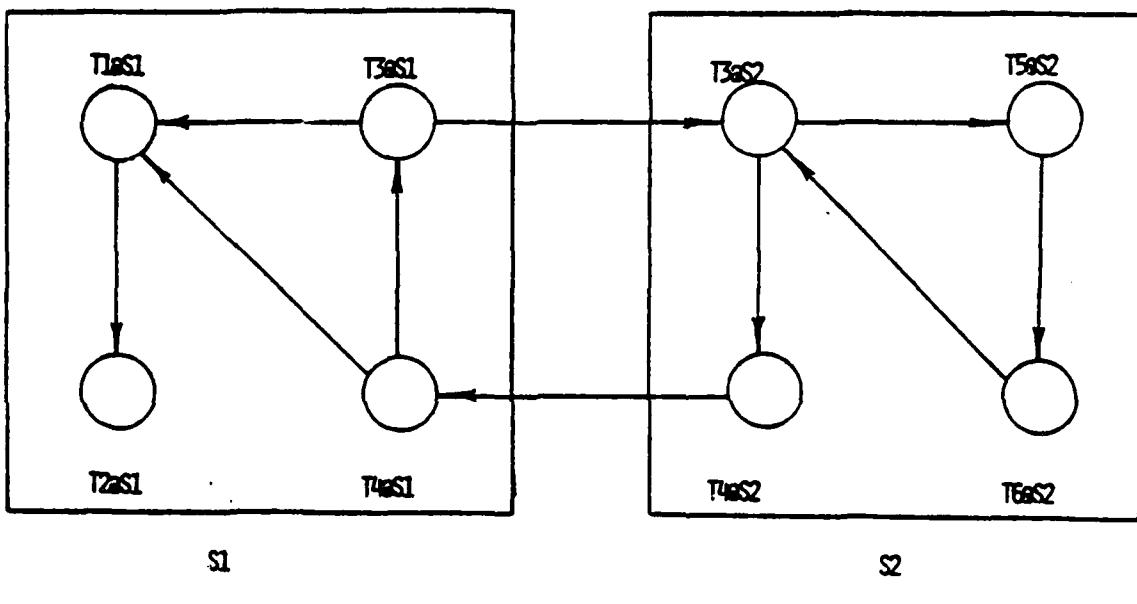


Figure 3.10 - A Transaction_Wait_For Graph for a network with two sites S1 and S2.

3.3.2 - Deadlock Detection Approaches

Deadlock detection involves building and maintaining the transaction_wait_for graph and searching for the existence of cycles in the graph. The graph has to be updated every time that a transaction changes state, either from active to blocked or vice versa. It should be noted however that new cycles can only potentially arise when a transaction is blocked. Deadlock resolution involves the selection of one or more transactions to be preempted in order for the cycle to be broken. The criteria used in this selection should try to minimize the penalty of preemption by any suitable metric. Examples of deadlock resolution methods are: preempt the transaction which owns less resources, preempt the transaction with the smallest roll back cost, preempt the transaction with the longest expected time to complete, preempt any transaction in the cycle. The examination of such criteria is beyond the scope of this dissertation.

A centralized approach for deadlock detection in distributed databases was suggested by Gray in [GRAY 78]. In this approach there is a centralized deadlock detector which is responsible for constructing a global graph equivalent to the transaction_wait_for graph considered here. The central deadlock detector builds the graph out of information received from all the participating sites in the network.

While the centralized method may be practical and efficient for local networks it may impose fairly large communications cost in geographically distributed systems. This observation stems from the fact that the central deadlock detector may be located very "far" from some of the sites in the network. As an example, we certainly do not want deadlocks which only involve resources located at some host computers located in Southern California to be detected at the East Coast. The hierarchical approach to deadlock detection presented in the next section lends itself to an optimization by which deadlocks can be detected by a site which is located as "close" as possible to the sites involved in the cycle.

3.3.3 - A Hierarchically Organized Locking and Deadlock Detection Protocol

Let the database, DB, be partitioned into a set of subdatabases DB_i's such that DB is the union of all the DB_i's and DB_i and DB_j are disjoint for i ≠ j. The locking and deadlock detection mechanism presented here has as its core a hierarchy of lock controllers which interact in a way to be explained in this section. First, we are going to distinguish between the controllers which are at the bottom-most level of the hierarchy, called leaf-controllers or LKs, and the non-leaf controllers or NLKs.

A leaf controller, LK_i , is assigned to each subdatabase DB_i . In the example shown in figure 3.11 we have three leaf controllers LK_1 , LK_2 and LK_3 and two non-leaf controllers NLK_0 and NLK_1 .

Each leaf controller, LK_i , maintains a transaction_wait_for graph, $TWF(LK_i)$. This graph contains all the nodes of the global TWF associated with transaction incarnations local to LK_i . In addition, two special types of nodes called output port nodes and input port nodes are introduced in the TWF of a leaf controller. These nodes are associated with the arcs of the global TWF which join incarnations in two distinct controllers and are defined as follows.

- a. a node in the TWF graph of LK_i is called an output port and denoted $O(LK_i, T)$ if the global TWF contains an outgoing arc from an incarnation of transaction T local to LK_i into a non local incarnation of T .
- b. a node in the transaction_wait_for graph of LK_i is called an input port and denoted $I(LK_i, T)$ in $TWF(LK_i)$ if in the global TWF there is an incoming arc into an incarnation of transaction T local to LK_i from a non local incarnation of T .

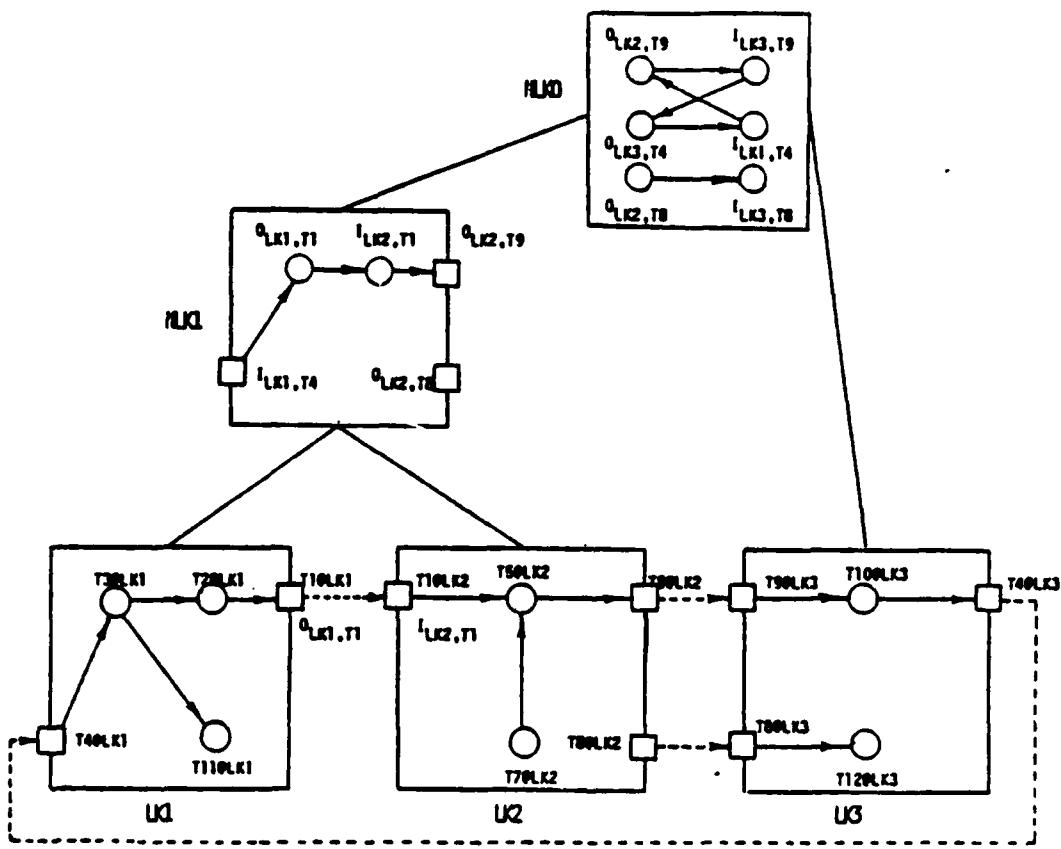


Figure 3.11 - A Hierarchy of Lock Controllers

Note that labels assigned to input and output ports are unique since there is only one transaction incarnation per transaction per site. In the example of figure 3.11, the output port $O(LK1, T1)$ and the input port $I(LK2, T1)$ correspond to an arc in the global TWF from $T1@LK1$ (the incarnation of $T1$ at $LK1$) to $T1@LK2$. The dashed lines indicate arcs in the global TWF. These arcs are represented explicitly at the upper levels of the hierarchy as will be explained below.

Non-leaf controllers maintain a graph called input-output-ports (IOP) graph. Nodes of an IOP are associated with input and output ports of leaf-controllers. We will refer to them as i-nodes and o-nodes respectively. Some of the i-nodes may be themselves input ports for the IOP and some of the o-nodes may be output ports for the IOP. The IOP for controller NLK_i , denoted $IOP(NLK_i)$, is defined by the following rules:

R1 - Arcs from i-nodes can go only to o-nodes and vice-versa.

R2 - There is an arc from o-node O_a to i-node I_b if O_a is an output port of a leaf controller in the subtree rooted at NLK_i and I_b is a corresponding input port of another leaf controller in the same subtree. In the example of figure 3.11, there is an arc from $O(LK1, T1)$ to $I(LK2, T1)$ in the IOP of NLK_1 since $O(LK1, T1)$ is an output port of

LK1 and I(LK2,T1) is its corresponding input port in LK2. LK1 and LK2 are in the subtree rooted by NLK1.

R3 - There is an arc from the i-node Ia to o-node Ob in IOP(NLK1) if there is a path from an input port Ia to an output port Ob of a son of NLKi. In the example of figure 3.11 there is an arc from I(LK1,T4) to O(LK2,T9) in NLK0 since in NLK1 there is a path between I(LK1,T4) and O(LK2,T9).

R4 - An input (output) port of IOP(NLK1) is also an input (output) port of a leaf controller in the subtree rooted by NLKi. In the example of figure 3.11 the input port of IOP(NLK1) is also an input port of LK1.

3.3.3.1 - Hierarchical Protocol - A Description

Before we describe the protocol let us define the lowest common ancestor between controllers K1, K2, ..., Kn, denoted lca(K1, K2, ..., Kn), as the common ancestor between them at the lowest level in the hierarchy (the root is at the highest level). Rules R1 through R3 below describe the hierarchical protocol.

RULE 1: (transaction incarnation T requests a local resource): The requested resource R is in the same subdatabase as the transaction incarnation T. Let LKi be the controller for resource R.

R1.1: [the resource cannot be granted] Let $\{T_1, T_2, \dots, T_k\}$ be the set of transactions which currently hold resource R. Add an arc from (T, LK_i) to (T_j, LK_i) for $j=1$ to k . Check the transaction_wait_for graph at LK_i for the existence of cycles.

R1.1.1: If cycles were formed then one or more local deadlocks have been detected and an appropriate action is required for deadlock resolution.

R1.1.2: The addition of the arcs mentioned in Rule R1.1 may have created one or more paths between input and output ports of LK_i . For each such path send the (input port, output port) pair which delimits the path to the father of LK_i .

RULE 2: (transaction T requests a non-local resource):
The requested resource R is in a different subdatabase from the previously requested resource. Therefore it has to be acquired by an incarnation of T local to R. Let LK_i be the controller for the previously requested resource and let LK_j be the controller for resource R. The incarnation of T at LK_i becomes blocked and waiting for a message from the incarnation of T at LK_j . The node (T, LK_i) is now an output port of the transaction_wait_for graph at LK_i and the node (T, LK_j) is an input port of the

`transaction_wait_for graph at LKj.`

R2.1: An arc from $O(LKi, T)$ to $I(LKj, T)$ is created in the IOP graph of the lowest common ancestor between LKi and LKj .

R2.2: An o-node labeled $O(LKi, T)$ is added to the IOP graph of each controller in the path between LKi and $\text{lca}(LKi, LKj)$. Each such o-node is also an output port of the corresponding IOP graph.

R2.3: An i-node labeled $I(LKj, T)$ is added to the IOP graph of each controller in the path between LKj and $\text{lca}(LKi, LKj)$. Each such i-node is also an input port of the corresponding IOP graph.

The protocol followed by a non-leaf controller, $NLKi$, is described by rule 3 below.

RULE 3: an arc is added to $\text{IOP}(NLKi)$.

R3.1: If a cycle is generated by the addition of the new arc then a global deadlock has been detected and an appropriate action is required to resolve it.

R3.2: If no cycle was generated check whether any input output port connection has been generated in IOP(NLK_i) and report the endpoints of any such connections to the father of NLK_i.

We have not yet mentioned how lock releases are reflected in the appropriate graphs in the hierarchy. Let each controller (LK or NLK) maintain a list of the i-o paths (i.e. the paths which connect input ports to output ports) in its graph. A possible representation for this list could be in the form of a bit matrix where each row corresponds to an i-o path and each column is associated with an arc in the graph. The value '1' in the i,j entry of this matrix indicates that arc j is in the i-o path i. An unlock operation causes an arc (maybe more) to be deleted from a TWF graph of a leaf controller. All the i-o paths (if any) which contained this arc are broken. This may be reported to the father of the LK. There, the arcs which represented the broken i-o paths will be used to find which i-o paths were broken in the non-leaf controller. This propagation continues up in the hierarchy until the deletion of an arc from a graph does not cause any i-o path to be broken.

The method described above for deadlock detection requires that non-leaf controllers be kept up to date continuously. Other variations can be used when appropriate. For

example, the information concerning connections between input ports and output ports can be sent periodically. For a sufficiently long period this would reduce the amount of traffic generated but may result in a deadlock existing for too long a period of time. Another method which is intermediate to continuous and periodic deadlock detection is to report connections between input and output ports after they have persisted longer than some threshold. Since, if a deadlock occurs the cycle persists until it is detected and relieved, this method will detect a deadlock after some delay. It appears that judicious choice for the threshold can result in less traffic being generated than with continuous checking and less delay in detecting a deadlock than with periodic checking.

3.3.3.2 - Hierarchical Protocol - Plausibility Argument

The hierarchical protocol has the following properties:

- a. deadlocks which involve resources of a single sub-database, DB_i, are detected by the formation of a cycle in the TWF of the leaf controller associated with DB_i.
- b. deadlocks which involve resources controlled by the leaf controllers LK₁, LK₂, ..., LK_i are detected by the formation of a cycle in the IOP graph

of the non-leaf controller which is the lowest common ancestor between the LK_i's.

Property **a** follows directly from Rule 1 of the protocol. The validity of property **b** is not so straightforward and will be shown to hold by the following theorems.

Let us introduce some notation first. Let the arcs which connect an i-node to an o-node in an input-output-ports graph be called $i \rightarrow o$ arcs and let $o \rightarrow i$ arcs be the ones which connect o-nodes to i-nodes. Let tree(K) be the subtree of the hierarchy rooted at controller K. Let us now extend the notation TWF(K) to indicate the subgraph of the global TWF obtained by considering only the resources controlled by all the LKs in tree(K).

Let us show that if a cycle is detected in the IOP graph of a non-leaf controller there is a deadlock.

THEOREM 3.9: If there is a cycle in the input-output-ports graph of a non-leaf controller NLK_i then there is a deadlock.

Proof: We need to prove that if there is a cycle C in the IOP graph of a non-leaf controller NLK_i there is an associated cycle C' in TWF(NLK_i). Let us show how to construct the cycle C'. Let C be a cycle in IOP(NLK_i). For the purpose of this construction let us label each $i \rightarrow o$ arc (I_j, O_j)

in C with the label Kj if Kj is an immediate son of NLKi such that the creation of an input output port connection in Kj caused the arc (Ij,Oj) to be created in IOP(NLKi) - see rules R1.1.2 and R3.2 of the protocol. Arcs of the type o->i in IOP(NLKi) will not be labeled. Using the notation (a,b,c) to indicate an arc labeled φ from node a to node b let the cycle C be $C = (I_1, O_1, K_1), (O_1, I_2, -), (I_2, O_2, K_2), \dots, (I_n, O_n, K_n), (O_n, I_1, -)$.

The repeated application of the operation 1 below will transform the cycle C into a cycle in which all the labels indicate leaf controllers.

Operation 1: If K_j in (I_j, O_j, K_j) is a non-leaf controller, replace (I_j, O_j, K_j) by a path connecting I_j to O_j in $IOP(K_j)$.

After this transformation, there is a path in the TWF of an LK in $tree(NLKi)$ associated with each $i \rightarrow o$ arc in C. There is also an arc between incarnations of transactions in the TWFs of distinct LKs associated with each $o \rightarrow i$ arc in C. More precisely, for each $i \rightarrow o$ arc, (I_i, O_i, K_i) , in C there is a path in $TWF(K_i)$ between the input port I_i and the output port O_i of $TWF(K_i)$. Now, by rule R2.1 of the protocol each $o \rightarrow i$ arc, $(O_i, I_{i+1}, -)$, in C connects two incarnations of the same transaction. These incarnations are local to leaf controllers in $tree(NLKi)$. Finally, the sum of all the paths and arcs thus obtained defines the cycle C' in

TWF(NLK_i).

In order to conclude the proof we must show that the arcs in the cycle C' defined above exist simultaneously and therefore C' is a deadlock cycle. Assume not. Then there is at least a pair of arcs T_i → T_j and T_m → T_n which did not appear simultaneously in C' and such that there is a path from T_j to T_m in TWF(NLK_i). Consider first the case in which T_i → T_j existed (i.e. appeared and disappeared) before T_m → T_n. Then, transaction T_j released the resource it was holding (which was needed by T_i). This implies that all the transactions in the path from T_j to T_m in TWF(NLK_i) must have had released at least one resource also. Since transactions are assumed to be two-phase, then none of them (including T_m) can issue any further lock requests. Therefore the arc T_m → T_n cannot exist. This contradicts our assumption and shows that it is not possible for T_i → T_j to have existed before T_m → T_n. Consider now the case in which T_m → T_n existed before T_i → T_j. This case is perfectly possible so long as these two arcs are not part of the same cycle otherwise one would have to conclude that every arc in the cycle existed before itself. Therefore, all the arcs in C' coexist in TWF(NLK_i) and C' represents a deadlock cycle and the theorem is proved.

We want to show now, that given a cycle in the global TWF there is a corresponding cycle in one of the controllers

in the hierarchy. In order to state this property more precisely some definitions are in order. Let the graph G' be called an arc-condensation or simply condensation of the graph G if the node set of G' is a subset of the node set of G and if (v_1, v_n) is an arc in G' then there is a path from node v_1 to node v_n in the graph G .

THEOREM 3.10: Given a cycle C in the global TWF there is an arc condensation of C in one and only one controller in the hierarchy. This controller is the lowest common ancestor of all the controllers which manage the resources in the cycle C .

Proof: Let C be a cycle in the global TWF which involves resources controlled by leaf controllers LK_1, LK_2, \dots, LK_n . Each of the n leaf controllers only know the portion of the cycle which contains resources local to the controller. Let us consider an initial condensation of the cycle C obtained by substituting every path in the TWF from an input port to an output port of a leaf controller by a single arc connecting these ports. For the purpose of this proof let us introduce a representation for a cycle which clearly illustrates how the knowledge about portions of the cycle is distributed throughout several controllers in the hierarchy. So, let a cycle be represented by a sequence of labeled arcs $(I_0, O_0, K_0), (I_1, O_1, K_1), \dots, (I_j, O_j, K_j), \dots, (I_{n-1}, O_{n-1}, K_{n-1})$ where (I_j, O_j, K_j) indicates that there is a

connection between the input port $I_{j'}$ and the output port O_j in the graph of the controller K_j . Also, the output port O_j is connected to the input port $I_{j+1} \pmod n$. Let $\text{father}(K)$ be the father of controller K in the hierarchy. The repeated application of operations 1 and 2 below gives us the desired condensed cycle C .

Operation 1: If all the labels in C are the same then stop. Substitute every maximal path P in C composed solely of arcs labeled with sons of a common controller K by a single arc having as endpoints the endpoints of P and having as label the common father controller K . This operation is illustrated in figure 3.12 and can be thought as a short circuit between consecutive brothers. Repeat operation 1 until all the labels in C are different and then do operation 2.

Operation 2: Find all the arcs in C which have as label a controller whose level in the hierarchy is the lowest of all the controllers which appear as labels in C . Substitute the label K_i in each of these arcs by $\text{father}(K_i)$. Do operation 1. Operation 2 is illustrated in figure 3.13.

In order to show the validity of operation 1 let us refer to figure 3.12 and consider the following observations:

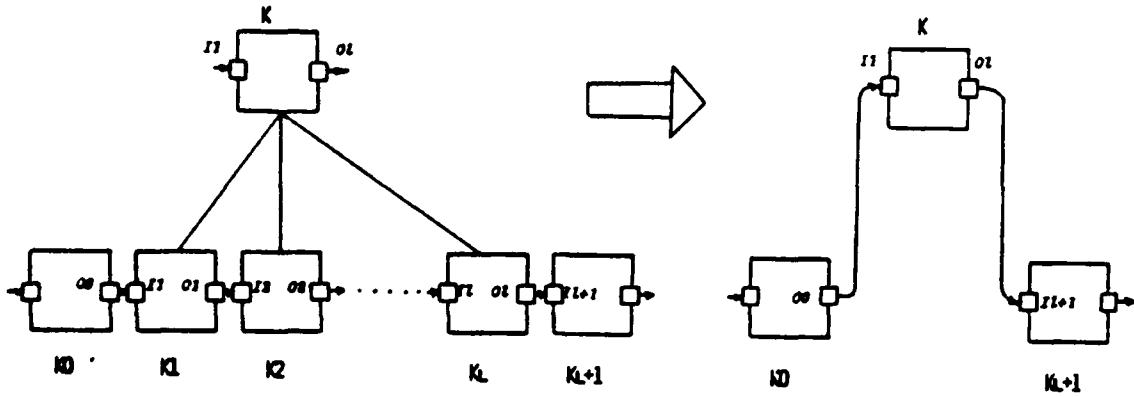


Figure 3.12 - Operation 1 in Theorem 3.10

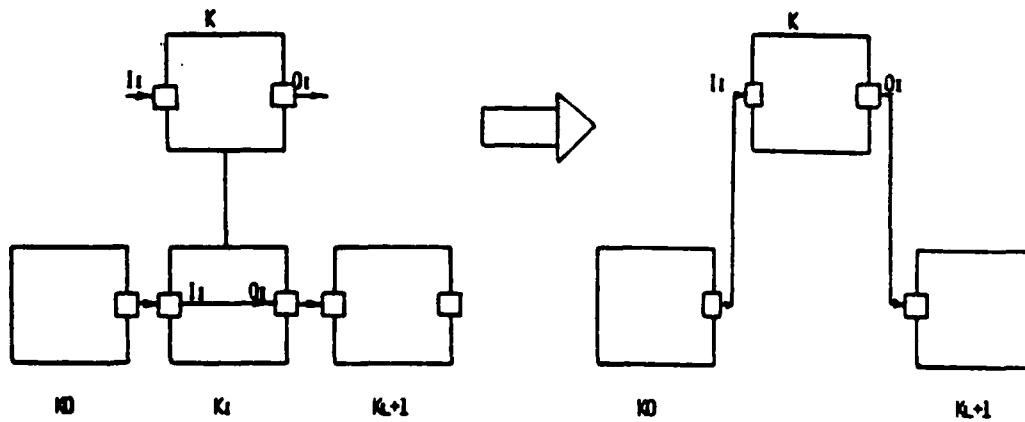


Figure 3.13 - Operation 2 in Theorem 3.10

- a. There is an arc from O_i to I_{i+1} for $i=1$ to $(l-1)$ in $IOP(K)$ since the controller K is the lowest common ancestor between K_1, K_2, \dots, K_l . See rule R2.1 of the protocol.
- b. There is an arc from I_i to O_i , for $i = 1$ to l in $IOP(K)$ since every time a connection between an input and an output port is created it is propagated to its father. See rules R1.1.2 and R3.2 of the protocol.
- c. Since the controller K_0 is not a son of controller K the input port I_1 is also an input port of K which is in the path between K_0 and $\text{lca}(K_1, K_0)$. See rule R2.3 of the protocol. The same observation applies to controller K_{l+1} and the output port O_l (see rule R2.2 of the protocol).

The reader should notice that operation 1 preserves the lowest common ancestor between all the controllers involved in the cycle.

The validity of operation 2 follows from observations b and c above. Notice that the choice of a controller at the lowest possible level to substitute during operation 2 preserves the lowest common ancestor between the controllers involved in the cycle. The reader can easily convince himself that this must be the case.

Therefore when all the labels are the same we have a representation of a condensed version of the original cycle. The common label is the name of the controller where the cycle is completely represented. Since operations 1 and 2 preserve the lowest common ancestor between the controllers involved in the cycle the original cycle will be represented in a condensed way in $\text{lca}(\text{LK}_1, \text{LK}_2, \dots, \text{LK}_n)$.

Theorems 1 and 2 together give us a necessary and sufficient condition for a deadlock to be detected in a distributed database involving resources at different sites. This result will be stated as the following theorem.

THEOREM 3.11: A necessary and sufficient condition for a deadlock involving resources in controllers $\text{LK}_1, \text{LK}_2, \dots, \text{LK}_n$ to exist is that there is a cycle in the IOP graph of $\text{lca}(\text{LK}_1, \text{LK}_2, \dots, \text{LK}_n)$. This cycle is a condensation of the corresponding cycle in the global TWF.

3.3.3.3 - Deadlock Resolution

As pointed out in an earlier section, we are not going to examine in this dissertation the criteria involved in optimal deadlock resolution since this is mainly a policy issue. This section discusses however the mechanisms which are necessary to allow the implementation of any such policy. The reader may have noticed that the condensed cycle in the IOP graph contains less information than the correspond-

ing cycle in the TWF graph. In particular, not all the transactions which participate in the cycle in the TWF graph appear in the IOP graph.

One way to compensate for this loss of information is to require that whenever an i-o arc is received by an NLK, the name of the controller which generated the arc be stored with the arc. Then, when an NLK detects a deadlock cycle it can send down the tree, to its appropriate sons, a message which will continue to propagate down (through the appropriate sons) until it reaches the leaves of the tree. At this point the LKs can report directly to the NLK which detected the deadlock all the necessary information to implement the desired policy for deadlock resolution.

Notice that the additional messages necessary to support the above described mechanism do not substantially increase the total communications cost of the protocol since they must only be sent when deadlocks are detected and not during normal operation.

Another, less flexible alternative, is to select the transaction to be preempted from those which appear in the IOP graph only. While no additional messages are required here it is likely that a non optimal choice will be taken in resolving the deadlock.

3.3.3.4 - Hierarchy Establishment

So far we have assumed the existence of the hierarchy without considering how it is established in the first place. The performance of the hierarchical protocol, in terms of the overhead message traffic incurred by it, can be minimized if the hierarchy is appropriately chosen. This choice should consider the pattern of the DB traffic with respect to the locality of access to a controller or group of controllers. For instance, assume that one is able to identify groups or clusters of leaf controllers such that a high percentage of the DB traffic involves controllers in the same cluster while very little traffic is of the inter cluster type. One possibility here would be to assign non-leaf controllers to each cluster and try to further cluster the non-leaf controllers or put all of them together under the same root.

The problem in general can be stated as follows. Given a set of leaf controllers, assigned to the nodes of a computer network, given the DB traffic pattern and given the cost of sending messages between every pair of nodes in the network, find a hierarchy which minimizes the total cost incurred in using the protocol.

There are clearly some heuristic rules which if applied to a given hierarchy result in another hierarchy of less cost. The general optimization problem, however, is the

subject of current research effort.

3.3.4 - A Distributed Locking and Deadlock Detection Protocol

A locking protocol which uses distributed control and a distributed deadlock detection mechanism is presented here. Before we describe the protocol, some definitions are in order. Each database site controls a set of resources. Transactions request resources by sending their requests to the controller of the resource. Each controller is responsible for:

1. processing lock and lock release requests for local resources. Requests may originate from any node in the network.
2. building a simplified version of the transaction_wait_for graph and detecting deadlocks. The TWF maintained by a controller is a subgraph of the global TWF.

For the purpose of stating this algorithm we will use a simplified version of the transaction_wait_for graph. The reader should be aware that this version is a redefinition of the graph used in the section on the hierarchical protocol although the same name for the graph is retained. In

this version there is no notion of transaction incarnations. Nodes are associated with transactions and there is a directed arc from transaction T' to transaction T'' if T' is blocked and must wait for T'' to release a resource (not necessarily a resource needed by T') before T' is able to proceed.

Some definitions are in order. A non-blocked transaction is a node in the `transaction_wait_for` graph with no outgoing arcs or a sink node. Let us now define blocking_set(T) as the set of all non-blocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T . This is the set of transactions which are ultimately blocking transaction T . The pair (T, T') is said to be a blocking pair of T if T' is in blocking_set(T).

The execution of a transaction can be described as follows. A transaction has a site of origin which is the site where the transaction entered the system. The transaction starts running at this site, performing local operations until operations on non-local data are necessary. Then, a lock request in the appropriate mode is built and sent to the controller for the requested resource. This controller will either accept or reject the lock, sending the reply to the site of origin of the transaction. If there are multiple copies of data, lock requests have to be sent to all

controllers which keep a copy of the data.

3.3.4.1 - Distributed Protocol - A Description

Let Sorig(T) be the site of origin of transaction T and let TWF(K) be the TWF graph at site Sk. The protocol is described by rules 1 and 2 given below as carried out by controller Sk. Let T be a transaction requesting resource R.

RULE 1: The resource R cannot be granted to T because it is being held by transactions T1, T2, ..., Tk.

Add an arc from transaction T to each of the transactions in the set {T1, T2, ..., Tk}. If the addition of these arcs caused a cycle to be formed in TWF(K) then a deadlock was detected and an appropriate action is required for its resolution. For each transaction T' in blocking_set(T) send the blocking pair (T,T') to Sorig(T) if Sorig(T) ≠ Sk and to Sorig(T') if Sorig(T') ≠ Sk.

RULE 2: A blocking pair (T,T') is received.

Add an arc from T to T' in TWF(K). If a cycle was formed then a deadlock exists and it must be resolved by an appropriate action. If T' is blocked and Sorig(T) ≠ Sk then for each transaction T'' in the blocking_set(T) send the blocking pair (T,T'') to Sorig(T'') if

$S_{\text{orig}}(T'') \neq S_k$.

Some comments are in order.

- a. the arcs of the `transaction_wait_for` graph considered here may represent one of two types of relationships between transactions, namely a direct wait and an indirect wait. Transaction T_1 is said to be waiting directly on T_2 if the resource needed by T_1 for its continued execution is being held by T_2 . A transaction T_1 is said to be waiting indirectly on T_k if there is a set of transactions T_2, T_3, \dots, T_{k-1} such that T_i is waiting directly on T_{i+1} for $i = 1$ to $k-1$.
- b. from the previous observation it can be seen that cycles in a TWF graph may be a condensation (in the sense defined in the section on the hierarchical protocol) of the cycle that would exist in the global `transaction_wait_for` graph for the whole system.

3.3.4.2 - Distributed Protocol - Plausibility Argument

That the protocol described in the previous section is able to detect all deadlocks is shown in the following theorem.

THEOREM 3.12: The above described protocol detects all possible deadlocks.

Proof: In order to show this result we will consider a global deadlock cycle, as shown in figure 3.14, and we will show that this cycle will appear in the TWF of the site of origin of at least one of the transaction in the cycle.

There are many orderings of resource requests that can lead to the same deadlock cycle. Each time a request is rejected the newly formed blocking pairs will increase the knowledge that some controllers have about the global graph. Therefore, the bigger the advance knowledge obtained when a request is rejected, more rapidly the deadlock will be detected. The ordering of resource requests used in this proof is such that controllers will get the minimum possible knowledge of the rest of the graph when a request is rejected. The reader should not have any problem in convincing himself that if the theorem holds for this ordering then it holds for any other. The chosen ordering is the following. Initially, transactions T_1 through T_{k-1} are blocked and each of the controllers at the site of origin of these transactions have the knowledge of a single transaction ahead in the cycle. At this point the `transaction_wait_for` graph at the site of origin of each transaction is shown below.

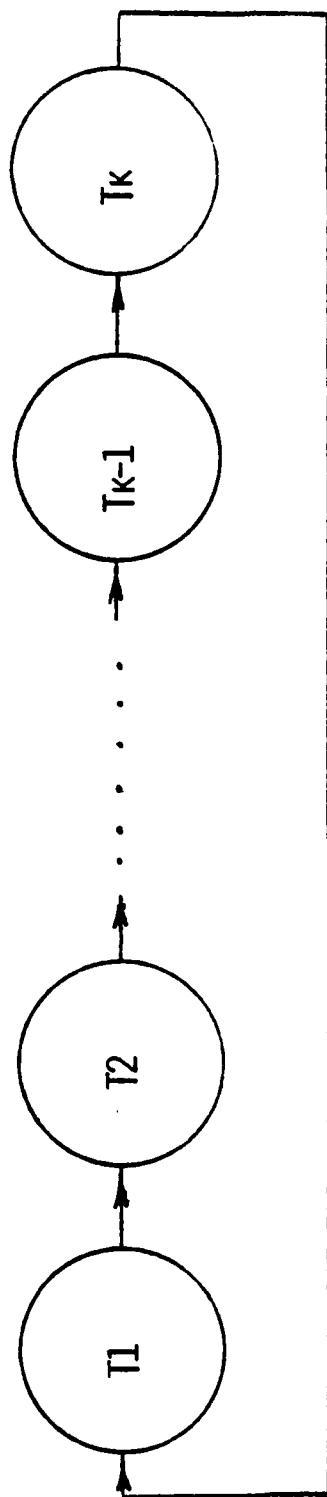


Figure 3.14 - Example of a Deadlock

```
Sorig(T1)      : T1 -> T2
Sorig(T2)      : T2 -> T3
.
.
.
Sorig(Tk-1)    : Tk-1 -> Tk
```

Now, when T_k makes a request and is blocked by T_1 , the blocking pair (T_k, T_2) will be sent to $Sorig(T_2)$ where a new blocking pair (T_k, T_3) is formed and sent to $Sorig(T_3)$. There the blocking pair (T_k, T_4) is formed and sent to $Sorig(T_4)$ and so on until the blocking pair (T_k, T_{k-1}) reaches $Sorig(T_{k-1})$. This causes the arc from T_k to T_{k-1} to be added to the TWF graph at that site. Since this graph already contained an arc from T_{k-1} to T_k a cycle is formed and the deadlock is detected.

3.4 - Related Work

This section describes several other approaches, suggested in the literature, for synchronization in distributed databases. The list of work is not intended to be exhaustive but it is rather representative of the spectrum of proposed solutions.

Time Stamps (Thomas)

The scheme suggested by Robert Thomas in [THOM 76] will be referred hereafter as the Base Variable Time Stamps or BVTS algorithm, for reasons which will become clear in the following description. The BVTS scheme requires that the

database be fully replicated at every site, although it was indicated in [THOM 77] that this condition can be relaxed. The following description considers, for simplicity, the fully redundant case.

Let us first examine how an update request is formed and how it is either accepted or rejected by the set of DBMPs (Data Base Manager Processes - there is one DBMP in every site). The sequence of messages necessary to form an update request is:

- an application program (AP) requests from any DBMP a set of variables, called base variables (BV), upon which the new values for the variables to be updated (update variables or UVs) are calculated.
- the DBMP replies to the AP by sending him the base variable values along with timestamps associated with them. A timestamp for a base variable is the time it was last modified.
- the AP calculates the values for the update variables (which must be a subset of the base variables) and sends to any DBMP an update request composed of:
 - i) a set of BVs and their originally obtained timestamps.
 - ii) a set of UVs with their new values.

When a DBMP receives an update request, it will be voted on by the set of DBMPs until the request is either globally accepted or rejected. The voting rules can be found in [THOMAS 76] but they basically amount to:

- voting OK on the request if its BV timestamps are current and if it does not conflict with a pending request. Two requests are said to conflict if the intersection of the set of UV variables of one with the set of BV variables of the other is not empty. A request is said to be pending if it has been voted OK but not globally accepted yet.
- voting REJECT if the BV timestamps are not current.

One REJECT vote is enough to globally reject the request. The majority of the DBMPs must vote OK on a request in order for it to be globally accepted. If a request is rejected it must be resubmitted.

In the BVTS scheme, update requests propagate in a daisy chain fashion from one DBMP to another. An expression for the delay D experienced by an update request from the moment the base variables are requested until the update request is finally accepted (probably after being resubmitted several times) can be obtained using a simple probabilistic model. The delay was found to be

$$D = T * [4 * p + qr * (p - 1) + qa] \quad (3.14)$$

where,

T = average message delay introduced by the network.

p = average number of times that the update request must be submitted until it is finally accepted.

qr = average number of DBMPs that vote on a request each time it is submitted and rejected.

qa = average number of DBMPs that vote on a request given that it is accepted.

Let us find D for the best case, which occurs when p = 1 (i.e., the request is accepted at the first time it is submitted) and qa = n/2 (since a majority consensus must be reached). In this case we have,

$$D_{best} = T * (4 + n/2) \quad (3.15)$$

The average communication cost, Cupdt, for the BVTS algorithm is

$$Cupdt = 2*M*[qa + (1+qr)*(p-1) + 1] \quad (3.16)$$

where M is the average communication cost per message and p, qr and qa are as defined before.

Again the best case occurs for p = 1 and qa = n/2 giving us

$$C_{best} = (n + 2) * M \quad (3.17)$$

We reproduce here, for convenience, the equivalent results we obtained for the CLC protocol.

$$D_{updt} = 2*T + 3*TMAX + W \quad (3.18)$$

$$C_{updt} = (3*n - 1)*M \quad (3.19)$$

As it can be seen, the average delay for an update in the BVTS protocol, even in the best case, grows linearly with n . Now, for the CLC protocol the delay, D_{updt} , given in (3.18) depends on $TMAX$ which for many network topologies of interest is not a direct function of the size of the network.

From (3.18) and (3.19) it can be seen that for fully replicated databases the CLC protocol has a higher communications cost than the best case for BVTS. On the other hand, since an update request may have to be submitted many times, in the BVTS scheme, it is possible that in some cases this situation is reversed.

An important limitation of the BVTS scheme is that it does not provide for value dependent locks. As shown by Eswaran in [ESWA 76], it is necessary to be able to specify the area of the database to be locked by some form of predicate calculus expression if we want to avoid the problems of

"phantom tuples".

The BVTS algorithm requires that a timestamp be stored in the database with every modifiable data item. This requirement causes a potential storage overhead, which can be reduced if one associates timestamps to groups of elementary data items instead of attaching a timestamp to each member of the group. For instance, one may associate timestamps to entire records of a file rather than associating a separate timestamp to each field of the record. While a coarse "time stamp granularity" implies in less storage overhead, it implies a higher update request rejection probability and consequently in a lower degree of concurrency and higher communications cost and delay. For the CLC protocol, the storage overhead lies in the fact that redundant copies of the LOCK table must be stored. However, since those tables only contain entries for active locks, we believe that in general, the storage overhead for the CLC protocol is smaller than for the BVTS algorithm.

Preprocessing (Rothnie and Bernstein)

The approach to update synchronization taken in SDD-1, as described in [ROTH 77 and BERN 77], takes advantage of the fact that different levels of synchronization may be required by different types of transactions. The SDD-1 update methodology prescribes four synchronization protocols, num-

bered from one to four, with increasing degrees of synchronization. If the types of transactions which will be run against the database are known in advance they can be grouped into disjoint classes of transactions and one of the four synchronization protocols can be assigned to each class. The mapping from transactions to transaction classes and from transaction classes to protocols can be constructed off line by the database administrator and compiled into tables which will be used at transaction run-time to select the appropriate protocol to be used.

SDD-1 consists of a collection of interconnected datamodules. Transactions in SDD-1 are executed in two stages:

1. a transaction is first executed at its site of origin. This execution, called an L action (for local), generates a list of updates to the database. The list of updates is timestamped with the time at which its L action was executed and broadcast to all the other datamodules.
2. the second stage is the processing of the update list at a remote module. This processing is called an U action.

L and U actions are assumed to be atomic. However, L and U actions of different transactions may be interleaved. The four synchronization protocols determine, in some sense, the degree to which L and U actions may be interleaved. The protocols can be basically described as follows:

Protocol 1:

This is the null protocol. In other words, this protocol provides no intermodule synchronization. It is the protocol to be used by transactions which do not interact, in some sense, with others.

Protocol 2:

This protocol is primarily used for retrieval transactions. A transaction t_2 is said to satisfy protocol 2 with respect to other transactions if it either sees all or none of the updates generated by transactions that precede it. A transaction is said to precede another if its L action was executed before the other, as indicated by the timestamp.

Protocol 3:

A transaction t_3 is said to satisfy protocol 3 with respect to other transactions if it sees all the updates to the database generated by transactions which precede it.

Protocol 4:

A transaction t_4 is said to satisfy protocol 4 with respect to other transactions if all the outstanding updates are executed before transaction t_4 executes its L action and if all the updates generated by t_4 are performed at other sites before any new transaction is introduced at those sites. This is the strongest of the four protocols and it is the one applied to all unanticipated transactions.

In the particular and interesting case in which types of transactions are known in advance, the SDD-1 solution may represent significant savings in communications cost and delay. In the general case, however the expensive protocol 4 would have to be used. For this more general case we advocate the use of the solution prescribed by the CLC protocol since it addresses, among other things, the issue of robustness and crash recovery not covered by the SDD-1 approach.

Sequential Ordering (Ellis)

In [ELLI 77] a solution to the update synchronization problem, called the ring structured solution, using a sequential propagation of synchronization and update messages is presented. The nodes of the network are ordered in a circular fashion and any node only receives messages from its predecessor and only sends messages to its successor.

An update request entered at a given node circulates through the set of database nodes. No updates are performed during this cycle. An update request u_1 can be temporarily removed from the cycle by any node j if there is an outstanding update request, u_2 , generated at node j of higher priority than u_1 and which conflicts with it. Previously removed requests are released by site j when all outstanding locally generated updates have been performed at all the copies of the database. The return of the update request to its source node after its first round trip indicates its acceptance by all the other nodes and generates a "perform update" message. This message circulates through all the nodes causing the update to be actually performed. No new updates can be introduced at site i if it knows of the existence of any outstanding update request which have not been completed yet.

The above solution preserves mutual and internal consistency by enforcing that updates are applied to all the copies of the database in the same order. This approach is somewhat restrictive and exhibits a low degree of concurrency. Communications cost for this solution are low ($2 \cdot n \cdot M$) at the expense of lengthy communication delays introduced by the serial propagation of messages. A formal verification technique has been carried out by Ellis [ELLI 77] to show the correctness of the ring structured solution. However, no indication was given as to how robustness and error

recovery issues are handled, if at all, by this solution.

Transaction Timestamping (Badal and Popek)

Another synchronization protocol, which uses timestamps on transactions and not on data items, was proposed by Badal and Popek in [BADA 78]. This protocol requires that read and write sets of a transaction be determined before the transaction is actually run. It is also assumed that once the read and write sets are determined, the sites which store these sets, called read and write sites respectively, can be determined by consulting an appropriate catalog.

The operation of the protocol can be described as follows. When a transaction enters the system, its read and write sets as well as its read and write sites are determined. A message called "SET UP MESSAGE" or SUM is sent to all read and write sites. This message carries among other things the transaction description and the transaction timestamp, assigned to it upon its entry to the system.

A "preferred read" site taken from the set of read sites of a transaction broadcasts to all sites in the network a "REQUEST" or REQ message. This message demands all recent update operations and SUM messages generated elsewhere but not yet received by the read sites of the transaction.

The requesting site (the preferred read site) waits until it receives an acknowledgment message (ACK) for the REQ message from all the sites in the network. These ACKs may contain updates which must be forwarded by the requesting site to each of the other read sites. These updates are placed into Read Command messages sent from the preferred read site to the remaining read sites. Therefore, the write part of a transaction "pyggibacks" onto the read part of subsequent transactions.

Upon receipt of a Read Command, a read site performs the updates in the order of their timestamps and then executes the read operation for the transaction in question. Therefore, since the REQ message was sent to all sites in the network to collect all outstanding updates (it is crucial that all the sites be involved at this step) and since the read sites apply all the updates before reading, it follows that no read operation takes place until all relevant write operations for the site of the read have been carried out. It is this property of the protocol which is important in preserving the consistency of the DB.

The performance of this protocol however, degrades seriously with the size of the network. In particular, the overhead communications cost to do an update under this protocol is approximately $(2*n - 1)$ messages, where n is the number of sites in the network and not the number of parti-

cipating sites in the transaction.

Other Approaches

Other proposed schemes, called primary copy strategies have been suggested in [BUNC 75, GRAP 76 and ALSB 76]. Alsberg in [ALSB 76] introduced some techniques aimed at providing a certain degree of resiliency to the single primary, multiple backup strategies discussed in [BUNC 75 and GRAP 76]. The primary copy scheme is primarily designed to maintain mutual consistency of databases subject to a somewhat limited types of update operations and it does not address explicitly the problem of internal consistency of a DDB subject to transactions of the more general type supported by the CLC protocol.

3.5 - Conclusion

The first part of this chapter outlines what we believe to be a fairly general solution to synchronization issues in distributed systems in the face of asynchronous unplanned failures. The algorithms and protocols for normal operation and recovery are robust with respect to the criteria set up at the beginning of section 3.2. We are unaware of any other synchronization protocols which simultaneously satisfy each of those requirements.

The work is primarily suitable for environments in which the cost, including delay, of sending messages is not high relative to the operations which are to be performed once locking is complete. Locally distributed systems often provide examples of such an environment. Geographically distributed networks also fall into this category if the amount of work to be performed after locking is significant relative to the communications cost.

The protocols are also best suited for usage behavior that cannot be directly characterized in advance. It is assumed that query and update activity will be largely ad hoc in nature - the more general case which has been receiving increasing attention in recent years.

The presentation of any substantial protocol would not be complete without an outline of a proof that the protocol is correct with respect to its desired properties. A significant portion of this document is therefore devoted to that purpose. In conclusion, these protocols should help demonstrate the practicality of integrated cooperation of activities in distributed systems.

The second part of this chapter presents two solutions to the problem of deadlock detection in distributed databases. The first solution consists of a hierarchy of lock controllers and is intended to achieve better performance, in terms of communications cost, than a centralized ap-

proach. For this purpose, the hierarchy should be established in a way such that deadlocks can be detected by a site which is located as "close" as possible to the sites involved in the deadlock. The problem of finding a hierarchy of lock controllers which minimizes the total cost incurred in using this protocol is the subject of current research. The design of a distributed protocol for deadlock detection was motivated by the desire to support reliable operation in environments subjects to failures.

Both protocols use a graph model to depict the current state of execution of all transactions in the system. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. The protocols presented here do not require that a global graph be built and maintained in order for deadlocks to be detected. An outline of the proof of the correct operation of the proposed protocols is included here.

The communications cost involved in using the solutions presented here depend on several factors such as database traffic pattern, distance between participating sites, hierarchy topology (for the hierarchical protocol only) and others. A detailed analysis of the performance characteristics of these protocols is the subject of further work.

CHAPTER 4

A FORMAL MODEL OF CRASH RECOVERY IN COMPUTER SYSTEMS

4.1 - Introduction

This chapter is concerned with issues of crash recovery in computer systems. In particular it addresses the following problem. Given a set of objects (processes, files, etc.), a set of snapshots for each one of them and the records of information flow between them, find the set of objects and their snapshots which should be used to restore the state of the system to a consistent state, once an error in any of the (potentially interacting) objects is detected. A formal model of crash recovery, in the form of a graph, is introduced in section 4. Several important results regarding the properties of the graph are stated and proved. Among these results we have:

- a. a specially defined cutset of the graph, called a crash set, identifies the objects and their snapshots which should be used for consistent state restoration.
- b. all the snapshots in any directed cycle of the graph are useless and can be therefore discarded since they can never participate in a crash set.

This chapter concludes with the presentation of an efficient algorithm to find the crash set given that a set of objects are known to have crashed.

4.2 - The Crash Recovery Problem

The strategy for error recovery which will be discussed in this chapter is called backward error recovery. This technique involves backing up to a previous state one or more system components (files, processes, etc.) when an error is detected. In order to illustrate what is involved in backward error recovery consider the following situation. Assume that a process, P1, crashes. Therefore, one would like to backup P1 to its most recent snapshot. However, all of the processes and files which have interacted with P1 after its most recent snapshot was taken must also be backed up. This fact may in turn force us to choose an earlier snapshot for P1, which in turn may bring more processes and files into the set of objects which have to be backed up, and so on. This domino effect as described in [RAND 77] continues until one is able to find a consistent set of snapshots for all processes and data objects involved. Therefore, backward error recovery requires:

- a. the existence of snapshots of files, processes and of all the other objects that one wishes to be able to recover.

b. the existence of an information flow detection mechanism between distinct processes and between processes and files, in order to allow the recovery software to determine which processes and or files should be backed up and to which point in time they should be backed up.

One way to graphically represent the relationship between snapshots of objects and information flow between them is shown in figure 4.1. In this representation, suggested by Randell et al. [RAND 77], each horizontal line represents a time axis for each object and the vertical lines represent information flow between objects. The vertical lines will be called interaction edges heretofore. The small vertical marks in the time axis of an object indicate the instants of time at which snapshots for that object were generated. We will use the term historical version instead of snapshot in the remaining of this work.

The problem of backward error recovery consists of finding a consistent set of historical versions of all the objects involved in a crash, restoring these objects and continuing if possible. In the example of figure 4.1, objects b, c and d have to be restored to the values of the historical versions b2, c2 and d1, respectively, if object b crashes.

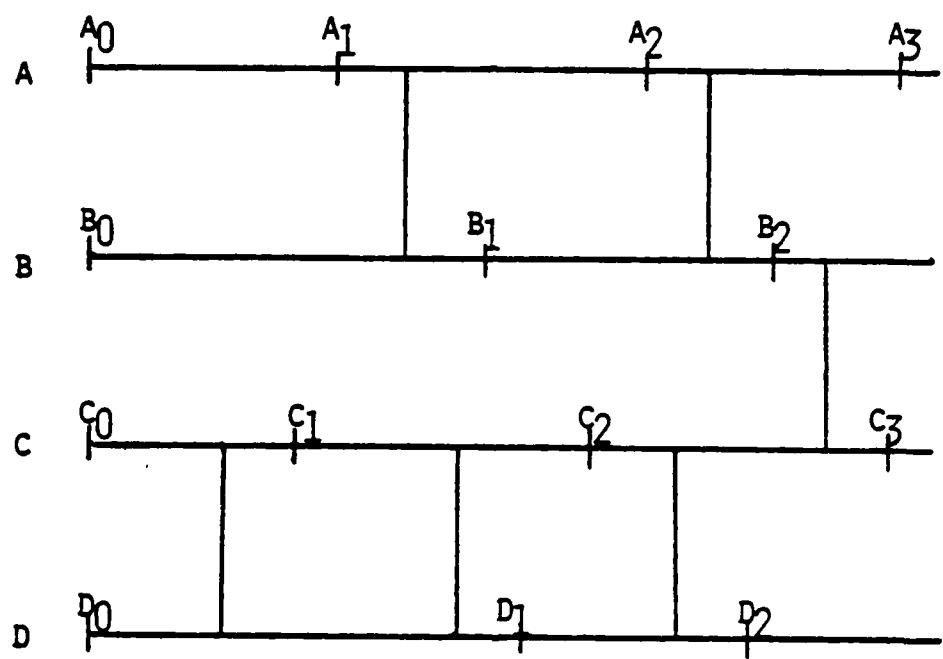


Figure 4.1 - Time Axis Diagram.

The statement of the problem will be made more precise as some definitions and a formal model for crash recovery are introduced in the following sections.

4.3 - Some Definitions and Properties

This section introduces several basic definitions and a necessary and sufficient condition for consistent state restoration. The state of a system is the collection of values of all the objects in the system.

Definition 1: (consistent state): A state of the system is said to be consistent if it is a state that the system might have reached through normal operation starting at a consistent state. The initial state of a system is consistent by definition.

Definition 2: (recovery set): Let S be a set of objects and let $H(S)$ be a set of historical versions of the objects in S such that there is exactly one historical version in $H(S)$ for each object in S . The set $H(S)$ is said to be a recovery set if it is the minimal* set of historical versions such that a consistent state of the system is obtained by restoring all the objects in S to their historical versions in

$H(S)$.

Since $H(S)$ is defined to be minimal it must not include historical versions of objects which need not be backed up in order to guarantee consistent state restoration. In the example of figure 4.1, $\{b_2, c_2, d_1\}$, $\{a_2, b_1, c_2, d_1\}$, $\{c_3\}$ and $\{d_2\}$ are some examples of recovery sets. Let us examine what are the necessary and sufficient conditions for an arbitrary set of historical versions to be a recovery set.

LEMMA 4.1: A necessary condition for an arbitrary set $H = \{h_1, h_2, \dots, h_k\}$ of historical versions of objects o_1, o_2, \dots, o_k to be a recovery set is that for every pair of historical versions in H there is no interaction edge which predates one and postdates the other.

Proof: We will assume that H is a recovery set and that lemma 4.1 is false and show that we reach a contradiction. Consider a pair of historical versions h_i and h_j of objects o_i and o_j respectively. Assume now the existence of an interaction edge after the historical version h_i was taken but before h_j was taken. Now, if objects o_i and o_j were restored to their values of h_i and h_j , respectively, then o_j would be in a state which would be a function of the existence of an interaction edge with o_i which never existed

* A set X is said to be minimal with respect to property P if no proper subset of X has property P .

as far as the value h_i for o_i is concerned. Therefore the values h_i and h_j for objects o_i and o_j respectively could have never coexisted, or equivalently, the system could have never reached such a state through normal operation. Since restoring o_i and o_j to h_i and h_j produces an inconsistent state the set H is not a recovery set. We have a contradiction which proves that lemma 4.1 must hold for the set H to be a recovery set.

A set of historical versions satisfying lemma 4.1 above will be called consistent. In the example of figure 4.1, the set $\{a_2, b_2\}$ cannot be a recovery set because lemma 4.1 is not satisfied.

Whenever one or more objects are known to be in error, it is necessary to find the set of all the objects which must also be backed up because they have interacted with the objects detected to be in error. A recovery set must therefore include historical versions for all such objects. The following definition makes this statement more precise.

Definition 3: (completeness of a set of historical versions): Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of historical versions of the associated objects in $S = \{o_1, o_2, \dots, o_k\}$. The set H is said to be complete if it is the minimal set such that there is no interaction edge e between an object o_i in S and an object x not in S such that the interaction

edge e postdates the historical version h_j of object o_j .

In light of the above definition we can state another necessary condition for a set of historical versions to be a recovery set.

LEMMA 4.2: A necessary condition for an arbitrary set H of historical versions to be a recovery set is that it be complete.

Proof: The argument is again by contradiction and it follows the same line as the previous one. If we assume that H is a recovery set and that the condition is false then there must be at least one object x which does not have an historical version in H such that there is an interaction edge between x and object o_i where the historical version h_i of o_i is in H . In this case, backing up o_i to h_i would generate an inconsistent state since object x would know of an interaction with o_i which never occurred as far as h_i is concerned. The minimality of H with respect to completeness follows from the fact that H is also minimal with respect to consistent state restoration. If H is not minimal with respect to completeness then there is at least one object represented in H which did not interact with any of the objects represented in H after the historical versions in H were taken. Therefore this object need not be backed up and H is not minimal with respect to consistent state restoration. This is a

contradiction and proves the point.

Given the example of figure 4.1, the set {b2, c2} cannot be a recovery set because it is not complete.

If a set of historical versions is both complete and satisfies lemma 4.1 then a consistent state is achieved if this set of historical versions is used to recover the associated objects.

THEOREM 4.1: A necessary and sufficient condition for an arbitrary set $H = \{h_1, h_2, \dots, h_k\}$ of historical versions of objects o_1, o_2, \dots, o_k to be a recovery set is that it be complete and consistent.

Proof: The necessity of the condition has been shown already, therefore we only need to show the sufficiency. Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of historical versions of objects o_1, o_2, \dots, o_k , such that H satisfies lemmas 4.1 and 4.2. The completeness of H guarantees that it is not necessary to backup any other object because objects o_1 through o_k are backed up. The completeness of H , by its minimality, also guarantees that no unnecessary object will be backed up, therefore guaranteeing that H is minimal with respect to consistent state restoration. Lemma 4.1 guarantees that the historical versions in H could have coexisted as values of objects o_1 through o_k . Therefore, the state obtained by backing up according to the recovery set H generates a con-

sistent state for the system.

There are several recovery sets to which the system can be backed up when a set of objects are detected to be in error. From these sets we want to select the one that minimizes the amount of backing up necessary. This recovery set is called the crash set.

Definition 4: (crash set of an object): Given an object x detected to be in error, the crash set of x , denoted $\text{crash_set}(x)$, is the latest possible recovery set which contains an historical version of the object. By latest possible we mean the one which contains the most recent possible historical version of the objects in the set.

From the example of figure 4.1, we have that $\text{crash_set}(a) = \{a_3\}$, $\text{crash_set}(b) = \{a_3, b_2, c_2, d_2\}$, $\text{crash_set}(c) = \{c_3\}$ and $\text{crash_set}(d) = \{d_2\}$.

The above definition of crash set of an object will be generalized to the notion of a crash set of a set of objects in section 4.7.1 after we introduce a formal model of crash recovery.

4.4 - Formal Model of Crash Recovery

This section introduces a formal model of crash recovery. This model is in the form of a graph called history graph. This graph has two types of branches, namely directed branches and undirected ones. There is a directed branch associated with each historical version of each object. Branches of this type are called hv-branches (for historical version). The nodes of the history graph have no semantics associated with them but they serve as endpoints of hv-branches. There is an undirected branch associated with each interaction edge. These branches are called ie-branches (for interaction edge). The history graph is defined by the following rules:

- a. For every object there is a directed path of hv-branches containing a branch for every historical version of that object. This path is such that if we traverse it in its proper direction we will encounter all the historical versions of that object in chronological order.
- b. Hv-branches are labeled with historical version names. Each node in the directed path associated with each object is labeled with the same label of the only directed branch incident out of it, if there is one. If the node has no outgoing direct-

ed branches then it is labeled with the name of the object followed by a star (*).

- c. Let v_i and v_j be two nodes in the graph associated with objects o_i and o_j respectively. Let h_i and h_j be the labels of the directed branches incident into v_i and v_j respectively. There is an interaction branch between nodes v_i and v_j if an interaction occurred between objects o_i and o_j after historical versions h_i and h_j were taken but before any other historical version for these objects was taken.
- d. We assume that there is always an initial historical version of each object which is taken before any interaction edge between the object and any other object occurs.

Let us add an additional node to the history graph, called the source node and labeled s . There is an undirected branch from this node to the starting node of the directed path of each object. This node is introduced to make the graph connected*.

* A graph with directed branches is said to be connected if in the underlying undirected version of it there is a path between every pair of nodes.

Figure 4.2 is the history graph for the situation illustrated in figure 4.1.

We will now examine some of the interesting properties of the history graph. These properties will be presented as theorems.

THEOREM 4.2: Let H be a recovery set. The set of hv-branches associated with the elements of H is a cutset* of the history graph.

Proof: We will first prove that the removal of the branches in H^{**} disconnects the history graph which is initially connected and then we will prove that H is minimal with respect to the property of being a disconnecting set. Before proceeding to the proof some definitions are in order.

Let G be the history graph. Let W be the set of objects associated with the historical versions in H . Let $G-H$ be the graph obtained from G by removing from G all the branches in H . Let G_x be the subgraph of $G-H$ induced by the set of nodes X defined as follows:

$$X = \{s\} \cup \{x \mid \text{there is a directed path in } G-H \text{ from } s \text{ to } x\}$$

* A cutset is a minimal collection of branches whose removal results in a non connected graph [FRAN 71].

** For the sake of simplicity we use the expression...branches in H ...instead of the precise expression...the set of hv-branches associated with the elements of H .

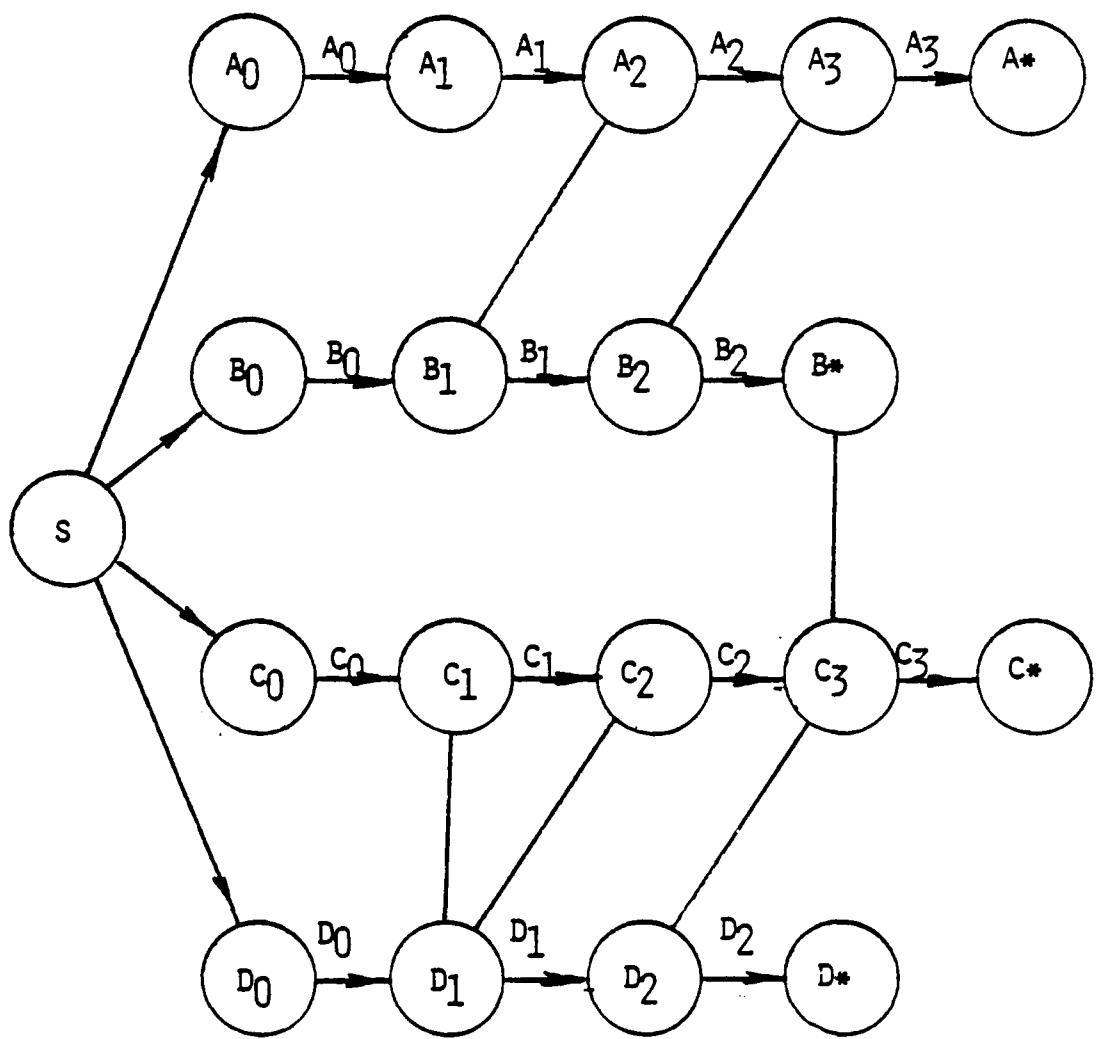


Figure 4.2 - History Graph.

Let Y be the set of nodes of G which are not in X and let G_Y be the subgraph of G induced in $G-H$ by the nodes in Y . Figure 4.3 illustrates the above definitions. Since the set H contains no historical versions of objects not in W then the directed path of such objects is totally contained in G_X , therefore the set Y only contains nodes associated with objects in W .

Let us prove the first part of the theorem. By definition of the set X , there is no directed branch in $G-H$ between a node in X and a node in Y otherwise the node in Y would be in X . It remains for us to prove that there are no undirected branches or ie-branches in $G-H$ between a node in X and a node in Y . There are two possible cases to consider:

Case 1: let e_1 be an ie-branch between nodes in the directed paths of objects in W .

The existence of such a branch indicates that there is an interaction edge that predates one historical version in H and postdates another. Lemma 4.1 is thus violated contradicting the assumption that H is a recovery set. Therefore an ie-branch such as e_1 cannot exist.

Case 2: let e_2 be an ie-branch between a node in Y and a node in the directed path of an object not in W .

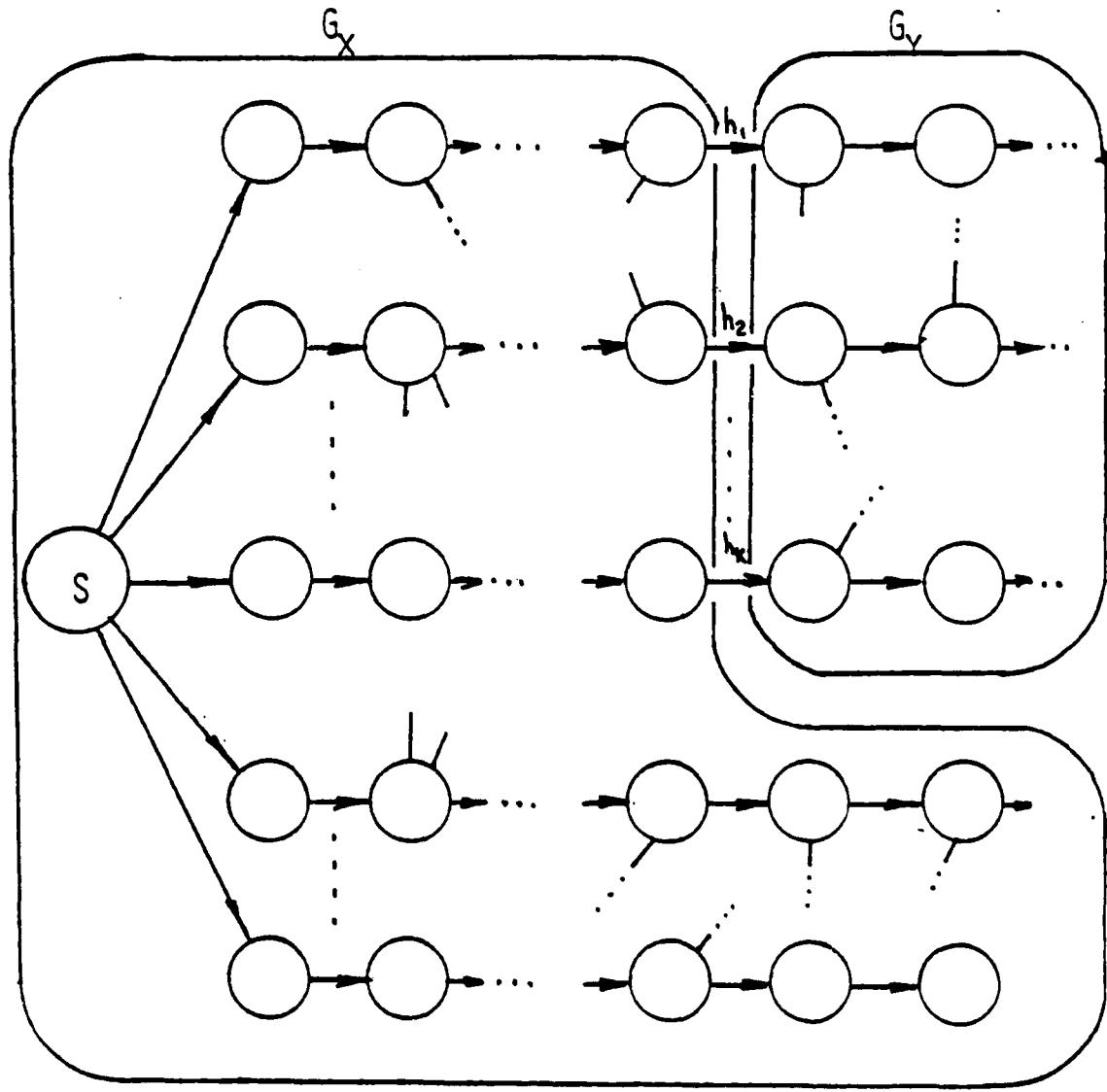


Figure 4.3 - Graphs G_x and G_y .

The existence of such a branch indicates that there is an interaction edge between an object w in W and an object z not in W which postdates the historical version of w . This is a violation of lemma 4.2 and contradicts the assumption that H is a recovery set. Therefore an ie-branch such as e_2 cannot exist.

Since the only branches that connect nodes in X to nodes in Y are the branches in H , the removal of these branches disconnects the graph G .

We will now prove that no proper subset of H disconnects G . Let us first show that G_y is connected. Assume that G_y is not connected. Then there is at least one object w in W which did not interact with any other object after its historical version in H was taken. Therefore, there is no object x in W such that there is an interaction edge between x and w which postdates the historical version of x in H . So, the set H is not complete which contradicts the assumption that H is a recovery set.

Now assume that H is not minimal then there is at least one branch h_i in H whose removal is not necessary to disconnect the graph. Therefore the graph $G - (H - \{h_i\})$ is disconnected. But since the graphs G_x and G_y are connected and since h_i connects a node in G_x to a node in G_y , the graph $G_x + G_y + \{h_i\}$ is connected which contradicts the assumption that H is not minimal and proves its minimality.

THEOREM 4.3: Let $H = \{h_1, h_2, \dots, h_k\}$ be a cutset of the history graph which contains no ie-branches and such that each of the branches in H corresponds to an historical version of a distinct object. Then the set of historical versions associated with H is a recovery set.

Proof: In order to prove this theorem we have to show that lemmas 4.1 and 4.2 hold, since these conditions together form a necessary and sufficient condition for H to be a recovery set. Let us define the sets X , Y and W and the graphs G , $G-H$, G_x and G_y as in the proof of theorem 4.2. Let us first prove that G_y is connected. Assume that G_y is not connected. That being the case, it has at least two components. Let C be one of the components and let $G_y - C$ be the graph formed by the remaining components. Since G is connected, there is at least one branch of G which goes from a node of G_y to a node of C . The remaining branches of H connect nodes of G_x to nodes of $G_y - C$. But removing the branches of H which connect G_x to C disconnects the graph and therefore H is not a minimal disconnecting set. This fact contradicts the assumption that H is a cutset and proves that G_y is connected. Since, by assumption, H is a cutset there is no ie-branch between a node in X and a node in Y otherwise removal of the branches in H would not disconnect G . Therefore, there is no interaction edge which predates an historical version in H and postdates another historical version in H . So, lemma 4.1 is satisfied. Also,

there is no interaction edge between an object w in W and an object z not in W which postdates the historical version of w in H . In order to prove that H is complete it only remains for us to show that there is no proper subset H_1 of H such that there is no interaction edge between an object x_1 in the set W_1 of objects associated with the historical versions in H_1 and an object z not in H_1 which postdates the historical version of x_1 . Assume that there is such a subset H_1 . Then, there is no ie-branch in G_y between nodes of G_y associated with the elements of W_1 and elements not in W_1 . Therefore G_y is disconnected which is a contradiction and proves that a subset such as H_1 cannot exist. Therefore lemmas 4.1 and 4.2 are satisfied and H is a recovery set.

The previous two theorems can be combined to yield the following result.

THEOREM 4.4: A set of historical versions $H = \{h_1, h_2, \dots, h_k\}$ is a recovery set if and only if the set of branches associated with the historical versions in H is a cutset of the history graph such that each branch in H corresponds to an historical version of a distinct object (i.e. is an hv-branch).

4.5 - Condensed History Graph

Since no cutset consisting only of hv-branches separates two nodes connected by an ie-branch we can collapse the graph by combining every pair of nodes connected by an ie-branch into a single "super-node". The label of a super-node is the union of the labels of the nodes of the component nodes. The set of branches incident into a super-node is the union of the sets of branches incident into the two component nodes. Similarly, the set of branches incident out of a super-node is the union of the sets of branches incident out of the two component nodes. The resulting graph after all collapsing has been done is called a condensed history graph. Of course the component nodes may themselves be supernodes which resulted from a previous collapsing operation. Since this graph only contains hv-branches we will simply use the term branch when referring to the branches of the condensed history graph. Figure 4.4 shows the condensed history graph for the history graph in figure 4.2.

The collapsing operation may create parallel* branches or self-loops** in the graph. Any cutset of a graph contains either all the parallel branches between two nodes or

* Two or more branches of a graph are said to be parallel if they have the same endpoints.

** A self-loop is a branch which is incident out of the same node into which it is incident.

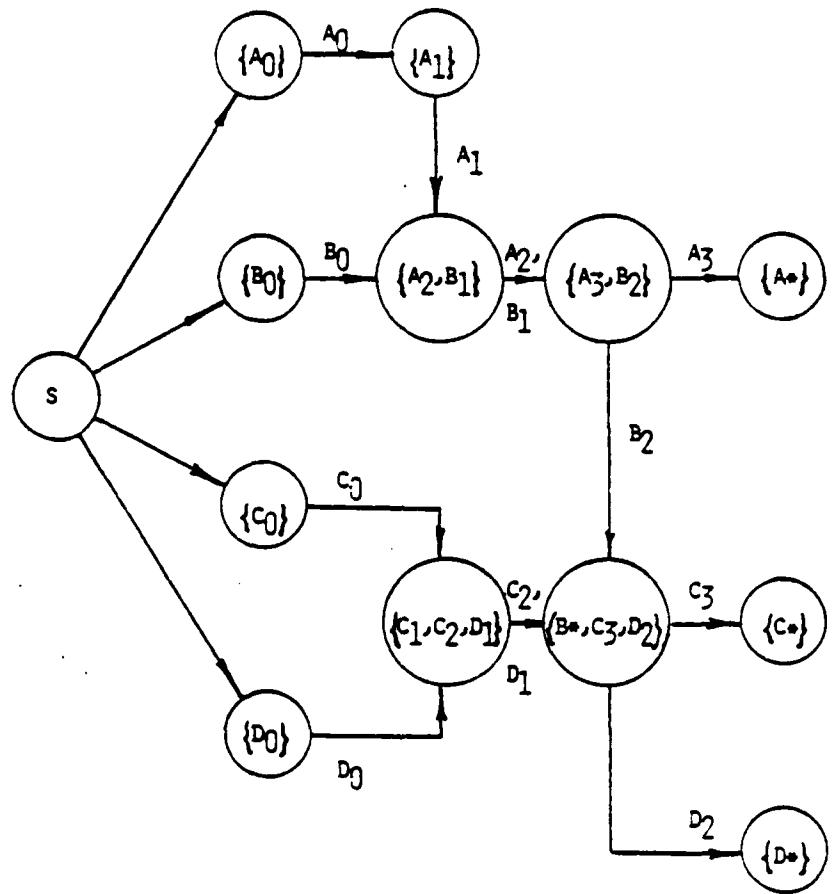


Figure 4.4 - Condensed History Graph.

none of them. Therefore we can combine all the parallel branches into a single branch labeled with the set of labels of the branches being combined. Also, since a self-loop will not appear in any cutset of a graph it can be eliminated from the graph. This fact has the implication that the historical version associated with the self-loop is useless as far as crash recovery is concerned. Consequently it need not be stored anymore and can be discarded. As it can be seen branches a_2 and b_1 have been combined into a single branch since, otherwise they would be parallel branches. Also, historical version c_1 disappeared, since otherwise it would be a self-loop of the graph. This result can be generalized as follows:

THEOREM 4.5: Let C be a directed cycle in the history graph in the sense that all hv-branches in the cycle are in the same direction if we traverse the cycle. The set of historical versions associated with the hv-branches in the cycle are useless and can be discarded.

Proof: We are going to prove that no hv-branch in C can belong to a cutset of the history graph containing only hv-branches and such that all the branches in the cutset are associated with distinct objects. It follows that such hv-branches cannot be part of any recovery set and represent useless historical versions which can be discarded. Let $H = \{h_1, h_2, \dots, h_k\}$ be a cutset of the history graph G such

that each branch in the cutset is an hv-branch and is associated with a distinct object. Let the sets X and Y and the graphs G_x and G_y be defined as in the proof of theorem 4.2. Let us first observe that all the branches in H are incident out of nodes in X and incident into nodes in Y. This follows from the definition of the set X, since the node s belongs to the set X and all the initial nodes of branches in H can be reached via a directed path from s. Now, let the hv-branch h_i belong to a directed cycle C and also to the cutset H. Let us now traverse the cycle C. If we go from X to Y by traversing the branch h_i we must eventually return to X by traversing a branch e. The edge e must be in the cutset H otherwise there is a path between every node in X and a node in Y which contains e and $G - H$ would not be disconnected. The branch e cannot be an hv-branch otherwise it would be incident into a node of X which is impossible as observed above. Also, the branch e cannot be an ie-branch since by assumption H only contains hv-branches. In conclusion, it is not possible for an hv-branch to belong to the directed cycle C and to the cutset H.

Let us observe that when we collapse nodes of a history graph into super-nodes we are essentially hiding all the ie-branches "inside" the super-nodes. So, a directed cycle in the history graph appears in the condensed history graph as a directed cycle containing only hv-branches. There is a

one to one correspondence between cycles in the history graph and its condensed version. We have therefore the following result.

THEOREM 4.6: Let C be a directed cycle* in the condensed history graph. The set of historical versions associated with the hv-branches in the cycle are useless and can be discarded.

Another interesting property of the collapsed history graph is that we do not need to store individual interaction edges anymore once their effect on the graph has been taken into account. The importance of this result can be better appreciated if one makes the following observations:

- a. If a crash destroys an historical version of an object then an earlier crash set may have to be used to recover from a crash of that object. However the ability to recover to a consistent state is not compromised by the accidental or intentional loss of an historical version.
- b. If any interaction edge is lost then correct crash recovery may not be possible any more since some of the crash sets may ignore the existence of interaction edges which actually occurred.

* We will consider a self-loop as a degenerate case of a cycle.

Observation b above implies that interaction edges must be reliably stored. This problem gets more complex because the amount of interaction edges generated per unit of time can be fairly large. One can expect thousands of interaction edges to be generated during one day of operation of a moderately loaded computer system. Therefore, if we had to store all of the interaction edges one would have to have a sort of file system for that purpose. This file system should be designed to be extremely reliable. We would also have the problem of garbage collecting interaction edges and retrieving them when needed for crash set calculation. Since interactions edges are crucial for correct crash recovery, extreme care has to be taken in deciding which ones can be discarded.

All of the above problems associated with having to store interaction edges have been solved by the introduction of the collapsed history graph.

A relationship between recovery sets and cutsets of condensed history graph, similar to the result of theorem 4.4, can also be proved. To this end we prove the following two lemmas.

LEMMA 4.3: Let $H = \{h_1, h_2, \dots\}$ be a history graph containing no cycles. If a branch in H is associated with a

AD-A186 683

SECURE DISTRIBUTED PROCESSING SYSTEMS(U) CALIFORNIA
UNIV LOS ANGELES SCHOOL OF ENGINEERING AND APPLIED
SCIENCE G J POPEK DEC 78 UCLA-ENG-7955

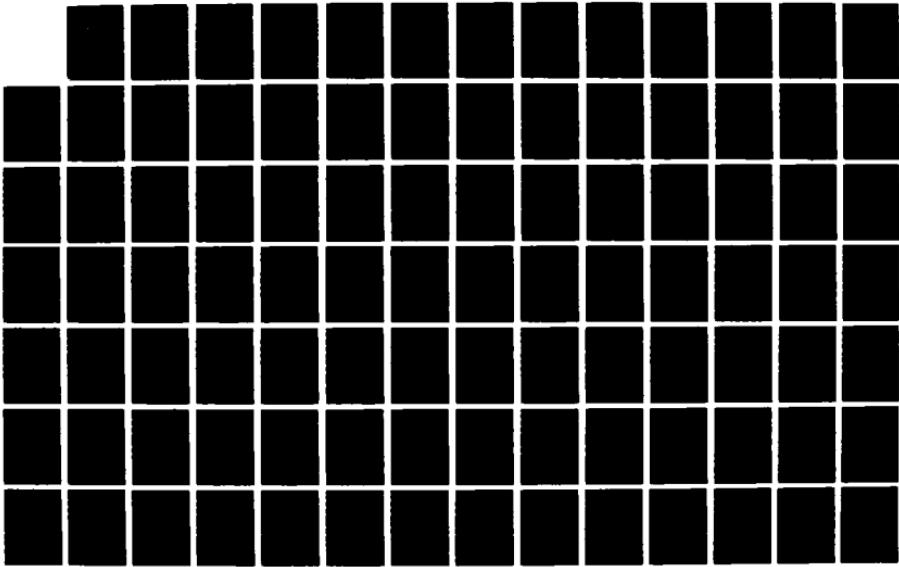
3/4

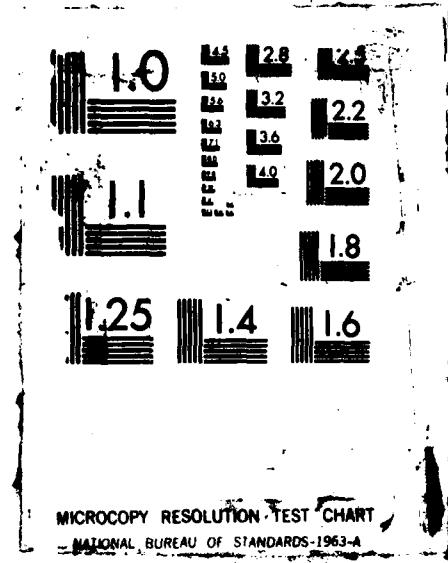
UNCLASSIFIED

MDA903-77-C-0211

F/G 12/7

NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963-A

distinct object. Let $H_c = \{c_1, c_2, \dots, c_m\}$ be the set of branches in the condensed version of G , G_c , associated with the branches in H . The set H_c is a cutset of the graph G_c such that the labels on the branches in H_c are associated with historical versions of distinct objects.

Proof: Let the sets X and Y and the graphs G , G_h , G_x and G_y be defined as in the proof of theorem 4.2. Let G_{xc} be the condensed version of G_x and let G_{yc} be the condensed version of G_y . Since H is a cutset with no ie-branches, there is no path containing only ie-branches between the endpoints of any branch in H . Therefore, the endpoints of such branches are not part of the same super-node in the graph G_c . Therefore, the associated branches in H_c are all incident out of nodes of G_{xc} and incident into nodes of G_{yc} .

Let us show now that the removal of the branches in H_c disconnects G_c by breaking all paths from G_{xc} to G_{yc} . Let us recall that G_x is connected by definition and that G_y was shown to be connected in the proof of theorem 4.3. Since the operation of collapsing nodes does not remove any branch, the graphs G_{xc} and G_{yc} are also connected. Since the branches in H_c are the only ones which connect nodes in G_{xc} to nodes in G_{yc} , the removal of such branches disconnects the graph G_c by breaking all the paths between nodes in G_{xc} and nodes in G_{yc} . This set is the minimal set with such a property, since any branch in H_c is part of a path

between G_{xc} and G_{yc} and must therefore be removed if G_c is to be disconnected.

LEMMA 4.4: Let H_c be a cutset of the condensed history graph G_c such that the labels on the branches in H_c are associated with historical versions of distinct objects. Let H be the set of branches of the history graph, G , associated with the branches in H_c . The set H is a cutset of the graph G with no ie-branches and such that the branches in H are associated with historical versions of distinct objects.

Proof: Let G_{xc} and G_{yc} be the two components into which the graph G_c is separated when the branches in H_c are removed from it. Let the graphs G_x and G_y be the uncondensed versions of the graphs G_{xc} and G_{yc} respectively. In order to obtain the uncondensed version of a condensed graph G_c we must replace every super-node in G_c by the connected subgraph in G , consisting of ie-branches only, which corresponds to the super-node. If G_c is connected so is G . Also, the endnodes of any ie-branch in G are combined into the same super-node of G_c . Therefore, there is no ie-branch connecting a node in G_x to a node in G_y . Otherwise, assume that there was such a branch. Its endpoints would be collapsed into the same super-node which would be both in G_{xc} and in G_{yc} . But this contradicts the assumption that G_{xc} and G_{yc} are two components of G_c . Therefore, all the branches connecting nodes in G_x to nodes in G_y are those in

H. It follows that the removal of these branches disconnects G. Since G_x and G_y are connected this set is also minimal. Otherwise assume that H is not minimal. Then there is at least one branch h_i in H whose removal is not necessary to disconnect the graph. Therefore $G-(H-\{h_i\})$ is disconnected. But since there is a path from every node in G_x to every node in G_y passing through h_i , the graph $G_x + G_y + \{h_i\}$ is connected. This fact is a contradiction and proves that H is minimal. Finally, the fact that the branches in H correspond to historical versions of distinct follows directly from the fact that the labels on the branches of H_c also correspond to historical versions of distinct objects. This concludes the proof.

Given lemmas 4.1 and 4.2 we can now state and prove the following result.

THEOREM 4.1: An arbitrary set $H = \{h_1, h_2, \dots, h_k\}$ of historical versions of distinct objects is a recovery set if and only if the set of branches associated with the historical versions in H is a cutset of the condensed history graph such that the labels on the branches of H correspond to historical versions of distinct objects.

Proof: The proof of this theorem is rather simple given theorem 4.4 and lemmas 4.1 and 4.2. If H is a recovery set, then by theorem 4.4 the set of branches associated to the

historical versions in H is a cutset of the history graph. But by lemma 4.3 the set of branches associated with the cutset in the history graph is a cutset in the condensed history graph. To show that the theorem is valid in the other direction we must use lemma 4.4 and theorem 4.4.

The history graph or the condensed history graph are dynamic in the sense that they change as new historical versions are generated, as interaction edges occur or as historical versions are intentionally discarded or lost due to crashes. The next section describes which operations have to be performed on the graph in order to model the above described events.

4.6 - Operations on the Graph

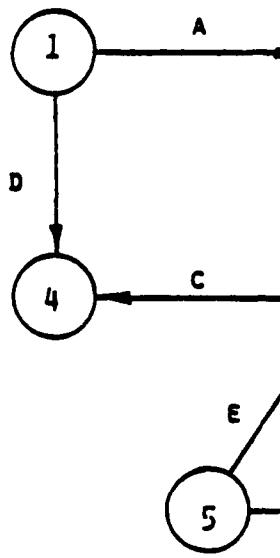
There are three operations that can be performed on the condensed history graph that are of interest to us, namely:

- a. addition of a new historical version
- b. addition of an interaction edge.
- c. intentional or accidental loss of an historical version.

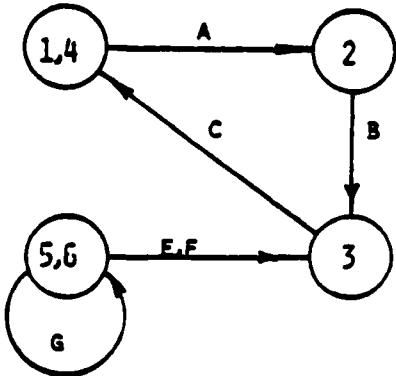
Adding a new historical version is a simple operation. The addition of an interaction edge causes two nodes of the

graph to be collapsed into a single node. The loss of an historical version causes the endnodes of that historical version to be collapsed into a single node. Each time that two nodes of a directed graph are collapsed there is a potential for the formation of directed cycles or self-loops. For instance, if in the graph of figure 4.5.a nodes 1 and 4 are collapsed into a single node we get a directed cycle as indicated in figure 4.5.b. Also, if nodes 5 and 6 are collapsed into a single node we get a self-loop.

This observation becomes important because, as shown in theorem 4.6, self-loops or branches which belong to directed cycles in the condensed history graph represent useless historical versions. These historical versions can be discarded because they cannot belong to any recovery set. It is necessary to introduce some notation before giving an algorithmic description of the operations mentioned above. Let label(v) be the label of a node v. Let $S:v \rightarrow w$ denote a directed branch incident out of node v and incident into node w and labeled with the set S. Let lastnode(x) be the node in the graph whose label contains x*. Let into(v) be the set of branches incident into node v and let out(v) be the set of branches incident out of v.



(4.5.a)



(4.5.b)

Figure 4.5 - Example of Generation of Cycles and Self-Loops (fig. 4.5.b) due to Collapsing of Nodes in Graph of fig. 4.5.a .

4.6.1 - Adding a new Historical Version

The following steps must be taken in order to add to the condensed history graph G_c the historical version x_i of object x .

S1 - Add branch $\{x_i\} : \text{lastnode}(x) \rightarrow w$, where w is a node not in G_c .

S2 - $\text{label}(\text{lastnode}(x)) \leftarrow (\text{label}(\text{lastnode}(x)) - \{x^*\}) \cup \{x_i\}$

S3 - $\text{label}(w) \leftarrow \{x^*\}$

4.6.2 - Adding an Interaction Edge and Discarding Historical Versions

These two operations are very similar in terms of what happens to the graph. They both require a collapse routine to collapse two nodes into a single node. In the case of an addition of an interaction edge the two nodes to be collapsed are the two lastnodes of the objects between which the interaction edge occurred. In the case of discarding an historical version, the nodes to be collapsed are the ones which are adjacent through the branch which represents the historical version to be discarded. Once two nodes of a directed graph are collapsed it is necessary to check for the existence of directed cycles or self-loops which might have been generated by the collapsing operation. These cy-

cles must be eliminated using the eliminate cycle routine.

We describe now each of the above operations.

Collapse Routine

The following steps must be taken in order to collapse nodes y and w in the condensed graph G_c .

S1 - Replace nodes v and w by a single node z such that

```
label(z) <- label(v) U label(w)  
into(z) <- into(v) U into(w)  
out(z) <- out(v) U out(w)
```

S2 - [check for parallel branches] Let

$$Y(y,z) = \{ S_i:y \rightarrow z \mid y \text{ is a node of } G_c \text{ and } y \neq z\}$$

and

$$X(y,z) = \{ S_i:z \rightarrow y \mid y \text{ is a node of } G_c \text{ and } y \neq z\}.$$

For all nodes y in G_c such that $|Y(y,z)| > 1$

replace all the branches in $Y(y,z)$

by the single branch $S:y \rightarrow z$

where S is the union of all the S_i 's.

For all nodes y in G_c such that $|X(y,z)| > 1$

replace all the branches in $X(y,z)$

by the single branch $S:z \rightarrow y$

where S is the union of all the S_i 's.

S3 - [check for self-loops] Let $D = \text{into}(z) \cap \text{out}(z)$.

If $D \neq \emptyset$ then discard all historical versions represented by the branches in D .

S4 - [check for directed cycles] GO TO the cycle_elimination routine.

Cycle elimination Routine

The cycle elimination routine is used to eliminate from the graph G_c all directed cycles. Elimination of such cycles is done by collapsing the endpoints of each branch in the cycle into a single node and deleting the branches from the graph. In other words, this operation discards all the historical versions represented by the branches in the cycle since they are useless. We are therefore interested in finding the set of all branches of G_c which belong to a directed cycle. But this is precisely the set of branches which belong to all strongly connected components of the graph. A strongly connected component of a graph G is a maximal subgraph of G such that there is a directed cycle containing every two pair of nodes in the component. A very efficient algorithm to find all the strongly connected components of a directed graph is given by Tarjan in [TARJ 72]. His algorithm requires $O(n,m)$ space and time*, where n is the number of nodes in the graph and m is the number of

* An algorithm is said to require $O(n,m)$ space and time if its space and time requirements are bounded by $k_1 n + k_2 m + k_3$ for some constants k_1 , k_2 and k_3 .

branches.

The cycle_elimination routine is then the following.

S1 - Find all strongly connected components of G_c .

S2 - For each strongly connected component collapse all its nodes into a single node and delete all of its branches.

4.7 - Crash Set Calculation

Let us first characterize a crash set of an object in terms of the condensed history graph. This characterization will lead to a simple and efficient algorithm for finding the crash set of an object. Some definitions are in order. Let $G^*(a)$ be a graph associated with object a such that the node set, V , of $G^*(a)$ is defined as

$$V = \{w \mid \text{there is a directed path from } \text{lastnode}(a) \\ \text{to } w \text{ in } G_c\}$$

The graph $G^*(a)$ can now be defined as being the subgraph of G_c induced by the set of nodes V . Figure 4.6 shows the graph $G^*(b)$ for the condensed history graph illustrated in figure 4.4. Let us observe that the graph $G^*(a)$ for any object a is a connected graph since, by definition, there is a directed path from $\text{lastnode}(a)$ to every node in the graph. We are now ready to prove the following result for con-

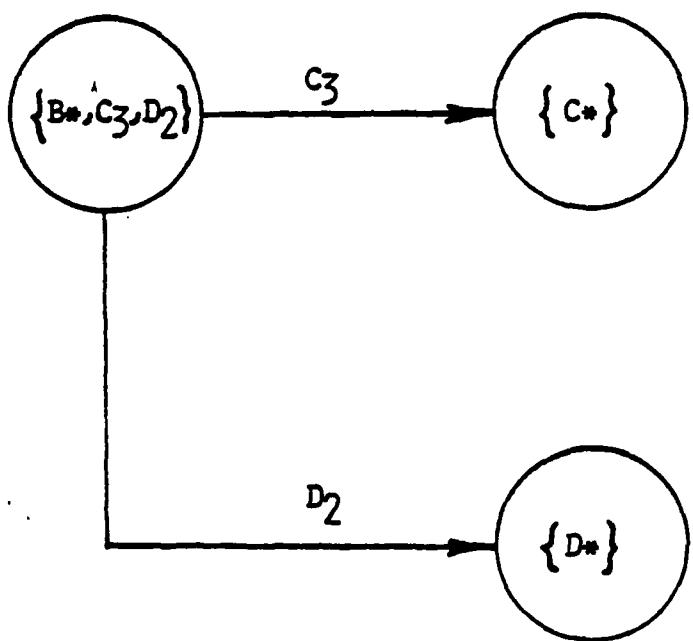


Figure 4.6 - Graph $G^*(b)$.

densed history graphs.

THEOREM 4.8: Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of branches of a condensed history graph G_c such that no branch in H is in $G^*(a)$ and all of them are incident into nodes of $G^*(a)$ for an object a . Then, $\text{crash_set}(a) = H$.

Proof: The set H is clearly a cutset of G_c since removal of the branches in H separates $G^*(a)$ from the rest of the graph G_c . Since G_c and $G^*(a)$ are connected the set H is the minimal set which disconnects G_c . We have to prove now that the branches in H correspond to historical versions of distinct objects. Assume that there are two branches h_i and h_j in H which correspond to historical versions of the same object. Assume also, without loss of generality, that h_i precedes h_j . Then there must be a directed path P in G_c which traverses h_i and h_j in this order. Let w be the node in $G^*(a)$ into which h_i is incident. But due to the existence of the path P , there is a directed path from $\text{lastnode}(a)$ to both endpoints of h_j . Therefore h_j is in $G^*(a)$ and cannot be in H . This fact shows that it is impossible for both h_i and h_j to belong to the set H .

So far we have proved that H is a cutset such that the branches in H correspond to historical versions of distinct objects. But, by theorem 4.7, this set is a recovery set.

It remains for us to show that H is the latest possible recovery set containing an historical version of object a . In other words, we have to show that there is no other recovery set H_1 which contains an historical version of a and that contains at least one historical version which postdates the corresponding historical version in H .

Let us make some definitions which are illustrated in figure 4.7. Let h_i and h_j be historical versions of the same object z and let h_i precede h_j . Assume that $h_i \in H$. Consider now a cutset H_1 of G_c such that the branches of H_1 are associated with historical versions of the same objects as the branches of H . Let the most recent historical version of object a be represented by the branch e and let $e \in H_1$. Let h_j and not h_i be in H_1 . Let the branches h_i and h_j be incident out of nodes y and w and incident into nodes x and z , respectively. Let G_1 and G_2 be the two components of $G - H_1$. Let the node s be in G_1 .

Assume that a cutset such as H_1 exists. Since $h_i \in H$, then node x is in $G^*(a)$. Therefore, there is a directed path P_1 in G_c from $\text{lastnode}(a)$ into node x . But since $\text{lastnode}(a)$ and node x are in separate components of $G - H_1$, then at least one branch of P_1 must be in H_1 . This branch must be incident out of a node of G_2 and incident into a node in G_1 . But this is not possible due to Lemma 4.5 proved below. Therefore it is not possible for a cutset

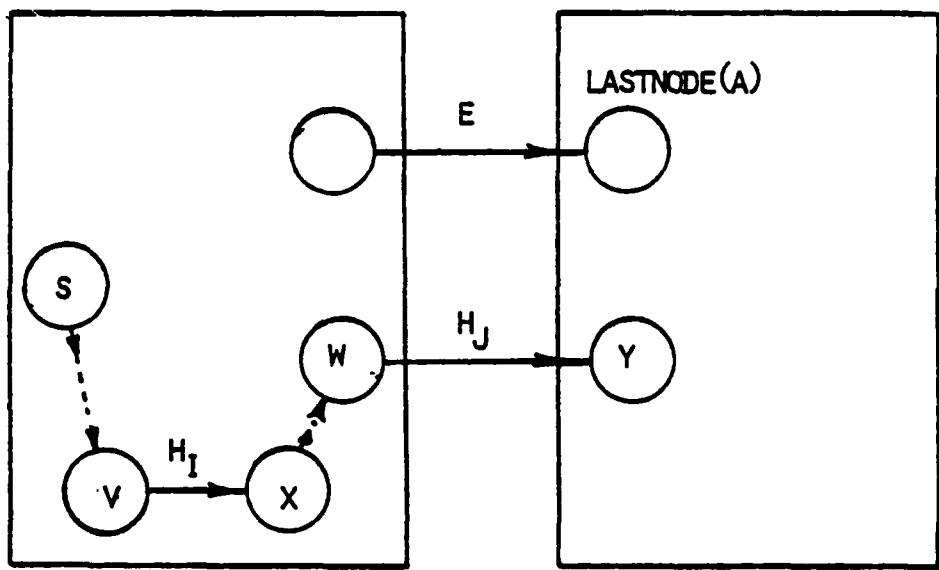


Figure 4.7 - Illustration for Definitions Made
in Theorem 4.8.

such as H_1 to exist, which proves statement ii) and completes the proof of the theorem.

LEMMA 4.5: Let G_c be a condensed history graph. Let H be a cutset of G_c associated with a recovery set and let G_1 and G_2 be the two components of $G-H$. Let node s be in G_1 . Then all the branches in H are incident out of nodes of G_1 and are incident into nodes of G_2 .

Proof: The proof is by contradiction. Assume that there is at least one branch $\in G-H$ which is incident out of a node of G_2 and is incident into a node of G_1 . Recall that there is a directed path in G_c associated with every object and starting at node s . But this path must contain at least two branches of H . One in each direction. Hence, there are at least two branches of H associated with historical versions of the same object. This contradicts the assumption that H is a recovery set and proves the lemma.

4.1.1 - Algorithm for Crash Set Calculation

The algorithm to calculate the crash set of an object a consists in finding the graph $G^*(a)$ and then finding the set of branches of the condensed history graph, G_c , which are not in $G^*(a)$ and that are incident into nodes of $G^*(a)$. The graph $G^*(a)$ can be easily found if we apply a graph traversal procedure such as the depth first search described in [TARJ 72], to the graph G_c starting at the node $\text{lastnode}(a)$.

This procedure will give us a spanning tree for the graph $G^*(a)$. The depth first search procedure is $O(n,m)$ and it visits each node of the graph exactly once assigning a number to it. Before we describe the algorithm let us define the following set associated to a given node v .

$$S(v,a) = \{v \rightarrow w \mid w \in G^*(a)\}$$

This is the set of all branches incident out of node v and incident into any node in $G^*(a)$. The algorithm is implemented by the procedure CRASHSET given below in Algol-like notation. It uses a depth first search procedure, called DFS, to traverse the graph $G^*(a)$ starting at node lastnode(a). As a result of the DFS procedure we get the array nodeset which indicates the node set of $G^*(a)$. A node j of G_c is in $G^*(a)$ if $\text{nodeset}[j] = \text{true}$. Also, after the end of the DFS procedure we have a collection of branches incident into nodes of $G^*(a)$. Some of these branches are in the graph $G^*(a)$ itself. These are the ones in $S(v,a)$ for nodes v in $G^*(a)$. Finally we take the union of all the sets of branches incident into nodes of $G^*(a)$. This union is equal to crash_set(a) by theorem 4.8.

Since the depth first search procedure is $O(n,m)$ the CRASHSET procedure is also $O(n,m)$.

Let us now generalize the notion of crash set of an object to that of crash set of a set of objects.

```

PROCEDURE CRASHSET(a);
BEGIN
    INTEGER i;
    BOOLEAN ARRAY nodeset[1:n]
    PROCEDURE DFS(v,u); COMMENT node u is
        the father of node v in the spanning tree of G*(a);
    BEGIN
        number[v] := i := i + 1;
        nodeset[v] := true ;
        FOR EACH x->v G into(v)
            DO S(x,a) := S(x,a) U {x->v};
        FOR w in the adjacency list of v DO
            IF w is not numbered THEN DFS(w,v);
    END;

    COMMENT initialization phase;
    i := 0 ; nodeset := false ; crash_set(a) = ∅ ;
    FOR EACH node x in Gc DO S(x,a) := ∅ ;

    COMMENT apply depth first search to find the
        set of nodes of Gc which belong to G*(a);
    DFS(lastnode(a),0);

    COMMENT crash_set(a) can now be found by taking
        the union of all the S(v,a) for nodes v
        not in G*(a);
    FOR j := 1 STEP 1 UNTIL n DO
        IF nodeset[j] = false
            THEN crash_set(a) := crash_set(a) U S(j,a);
    END;

```

Definition 5: (crash set of a set of objects): Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of objects. The crash set of the set X , denoted $\text{crash_set}(X)$, is the latest possible recovery set which contains an historical version of each object in X .

Before we state and prove the theorem which indicates how to calculate the crash set of a set of objects let us generalize the definition of the graph $G^*(a)$. Let $X = \{x_1, x_2, \dots, x_m\}$ be a set of objects. Let us define the graph $G^*(X)$ as the subgraph of G_c induced by the nodes in the set

V defined as

$$V(X) = \{v \mid v \text{ is in } G^*(x_i) \text{ for some } x_i \in X\}.$$

THEOREM 4.9: Let $X = \{x_1, x_2, \dots, x_m\}$ be a set of objects. Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of branches of a condensed history graph G_c defined as

$$H = \{v \rightarrow w \mid w \in G^*(X) \text{ and } v \notin G^*(X)\}.$$

Then, $\text{crash_set}(X) = H$.

Proof: The proof of this theorem follows much the same line as that of theorem 4.8. The set H is clearly a disconnecting set of G_c since removal of the branches in H separates $G^*(X)$ from the rest of the graph G_c . Since G_c and $G^*(X)$ are connected the set H is the minimal set which disconnects G_c . We have to prove now that the branches in H correspond to historical versions of distinct objects. Assume that there are two branches h_i and h_j in H which correspond to historical versions of the same object. Assume also, without loss of generality, that h_i precedes h_j . Then there must be a directed path P in G_c which traverses h_i and h_j in this order. Let w be the node in $G^*(X)$ into which h_i is incident. But w is in $G^*(x_i)$ for at least one $i=1, \dots, m$. Then, by definition of $G^*(x_i)$, there is a directed path from $\text{lastnode}(x_i)$ to w which concatenated with the portion of the path P starting at w implies that there is a directed path

from $\text{lastnode}(x_i)$ which includes h_j . Hence, h_j is in $G^*(x_i)$ and therefore in $G^*(X)$. Therefore h_j cannot be in H . This fact shows that it is impossible for both h_i and h_j to be in the set H .

So far, we have proved that H is a recovery set. Let us prove now that H is the latest possible recovery set which contains an historical version of each object in X . In other words, we want to show that there is no other recovery set H_1 which contains historical versions of each object in X and that contains at least one historical version which postdates the corresponding historical version in H . Let h_i and h_j be historical versions of the same object z and let h_i precede h_j . Assume that $h_i \in H$. Consider now a cutset H_1 of G_c such that the branches of H_1 are associated with historical versions of the same objects as the branches of H . Let h_j and not h_i be in H_1 . Let the branches h_i and h_j be incident out of nodes y and y' and incident into nodes x and x' , respectively. Let G_1 and G_2 be the two components of $G - H_1$. Let the node s be in G_1 .

Assume that a cutset such as H_1 exists. Since $h_i \in H$, then node x is in $G^*(X)$. Now, since node x is in $G^*(x_i)$ for some $i = 1, \dots, m$ then there is a directed path P_1 in G_c from $\text{lastnode}(x_i)$ into node x . But since $\text{lastnode}(x_i)$ and node x are in separate components of $G - H_1$, then at least one branch of P_1 must be in H_1 . This branch must be incident

out of a node of G_2 and incident into a node in G_1 . But this is not possible due to Lemma 4.5. Hence, a cutset such as H_1 cannot exist. This fact completes the proof of the theorem.

Given the result of theorem 4.9 we are now in a position to give the algorithm to find $\text{crash_set}(X)$. The algorithm consists of finding the graph $G^*(X)$ and then finding the set of branches of the condensed history graph, G_c , which are not in $G^*(X)$ and that are incident into nodes of $G^*(X)$. The graph $G^*(X)$ can be efficiently found by multiple applications of a depth first search procedure. One application starting at $\text{lastnode}(x)$ for each object $x \in X$. Although the algorithm uses multiple applications of the depth first search procedure, each node of the graph G_c will still be visited once since the algorithm preserves the node numbers assigned by each application of the DFS procedure. Also, if $G^*(x_i)$ is a subgraph of $G^*(x_j)$ for $x_i \in X$ and $x_j \in X$ the algorithm will not invoke DFS for $\text{lastnode}(x_i)$ if DFS was already invoked for $\text{lastnode}(x_j)$. Before we describe the algorithm let us define the set $S(v, X)$ of all branches incident out of node v and incident into any node of $G^*(X)$.

$$S(v, X) = \{v \rightarrow w \mid w \in G^*(X)\}$$

The algorithm which is described below in Algol-like notation is $O(n, m)$ since the DFS procedure is $O(n, m)$.

```

PROCEDURE CRASHSET(X);
BEGIN
    COMMENT nodeset[j] = true if node j is in G*(X)
          last[j] = true if node j is lastnode
                      for any object
          processed[j] = true if node j is numbered &
                           last[j] = true. ;
    INTEGER i;
    BOOLEAN ARRAY nodeset[1:n], last[1:n], processed[1:n];
    PROCEDURE DFS(v,u);
    BEGIN
        number[v] := i := i + 1;
        nodeset[v] := true ;
        FOR EACH y -> v G into(v)
            DO S(y,X) := S(y,X) U {y -> v};
        FOR w in the adjacency list of v DO
            IF w is not numbered THEN DFS(w,v);
        END;

    COMMENT initialization phase;
    i := 0; crash_set(X) := Ø ;
    FOR EACH node y in Gc DO S(y,X) := Ø ;
    FOR j:= 1 STEP 1 UNTIL n DO
        processed[j] := nodeset[j] := false ;

    COMMENT apply depth first search to find G*(x)
          for each object x G X. ;
    FOR EACH object x G X DO
        IF processed[lastnode(x)] = false
        THEN DFS(lastnode(x),0);

    COMMENT crash_set(X) can now be obtained by taking
          the union of all S(y,X) for nodes y G G*(X).
    FOR j := 1 STEP 1 UNTIL n DO
        IF nodeset[j] = false
        THEN crash_set(X) := crash_set(X) U S(j,X);
END

```

4.8 - Crash Recovery of Very Large Databases

The standard technique used to deal with errors in databases consists of periodically dumping the database, building a log of modifications to the DB since the last dump took place so that when an error is detected the data-

base can be reloaded from the dump and the log processed against it. Dumping a database can be either done with the database off line, i.e. by making it unaccessible to the users, or it can be made dynamically as suggested by Rosenkrantz in [ROSE 78]. Reloading the entire database from a dump or even reloading only the portion of it which was modified since the last dump took place is unacceptable in very large databases. As an example, consider the database which contains data gathered by the U.S. Census. It is estimated that the size of this database for the 1980 Census will be of the order of one Gigabyte (10^{12} bytes) [SHOS 78]. Let us assume that an error in the database was detected and let us also assume that on the average, ten percent of the database was modified since the last dump was taken. With state-of-the-art technology, typical data transfer rates for secondary storage devices are in the order of a few Megabytes/sec. Therefore, it would take of the order of one day to reload the portion of the database which was updated since the last dump took place.

Alternatively, one can make use of the model developped in this chapter in the following way. Partition the database into a number of logical subdatabases. Each such subdatabase is an object in our crash recovery model. An interaction edge is said to occur between two objects if an update transaction involving the two subdatabases they represent was issued by the user. Whenever one or more sub-

databases are detected to be in error we can do crash set calculation as described in the previous section in order to determine which portions of the database should be reloaded. As we see, this approach tries to make an assessment of the extent of the damage therefore allowing us to reload smaller amounts of data than with the previous technique.

Note that the choice of the partitions will determine to which degree one is able to reduce the portion of the database to be reloaded. Theoretically, if one partitions the DB in a way that each subdatabase is the smallest updatable data unit, then the crash set would give us exactly the portion of the database that was affected by the error. Such a fine grain is not feasible in practice and we will typically have subdatabases of much coarser grain. Typical examples could be whole relations or domains of a relation. At any rate, one should try to group in the same subdatabase, portions of the DB which tend to appear together in update requests.

4.9 - Conclusion

The condensed history graph introduced in this chapter as a formal model of crash recovery is important because of the following aspects:

1. once the effect of the information flow between

two objects has been taken into account in the graph, a record of it need not be stored anymore for crash recovery purposes. This fact is extremely important not only because it minimizes the storage requirements for the system but because the information flow pattern is crucial for correct crash recovery. Therefore if records of it needed to be stored, extremely reliable mechanisms should be provided to store them.

- ii. all the snapshots or historical versions which are useless because they cannot participate in any crash set can be detected to be so. Therefore they can be discarded.
- iii. crash set calculation can be efficiently done.

CHAPTER 5
CRASH RECOVERY IN DISTRIBUTED SYSTEMS

5.1 - Introduction

In the previous chapter we showed how to efficiently calculate the crash set for a set of objects given the condensed history graph G_c . The same approach cannot be used anymore in a distributed environment since we cannot assume that there is a single complete version of the graph G_c and moreover we cannot assume that the graph is being updated continuously to reflect events such as generation of new historical versions, occurrence of interaction edges and the like. The existence of a single centralized version of the graph is ruled out due to both reliability considerations and due to the fact that maintaining such a graph up to date would require that all purely local interaction edges be sent to the site which kept the complete version of the graph G_c . This approach implies in a message traffic overhead which is unacceptable. Instead of having a centralized complete version of the graph G_c we are going to assume that the global graph is partitioned into local subgraphs distributed among the several sites of the network. In the following section we will describe the criteria used in partitioning the graph into subgraphs.

Given that we have several portions of the global graph we need a protocol to update these copies. This protocol should be simple and exhibit as low as possible communications cost. This protocol will be described in section 5.3.2. We also need an algorithm which allows us to do crash set calculation in a distributed system. Crash set calculation in a distributed system should be distributed. Since the global graph is changing with time, performing crash set calculation at a central site may be done in one of the two following ways. The first one requires us to "freeze" the whole world, assemble the graph at one site and do crash set calculation as prescribed in the previous chapter. The second approach requires that the graph be assembled at one site and that every local modification to the graph be sent to the central site as soon as it is generated. This second approach would require a bandwidth of the order of the speed of light. A distributed algorithm to calculate the crash set is given in section 5.4.

5.2 - Partitioning the Graph

Let $SG(x)$ be defined as the subgraph associated with an object x , induced in the global graph G_c by the set of nodes associated with x plus all the branches incident into or out of these nodes. Let us also assume that each object x has a current location, denoted location(x), associated with it. A natural way to define location(x) is as follows:

1. If the object x is a process then $\text{location}(x)$ is the site where the process is currently running.
2. If the object x is a file then $\text{location}(x)$ is the site where the file is being currently accessed for update or the site where it was last accessed for update if the file is only being used for read.

The following criterium is used to partition the global graph G_c into subgraphs:

\forall object x , $SG(x)$ is located at $\text{location}(x)$.

As a consequence of the above criterium it follows that the subgraphs associated with processes running at the same site are all located at that site. Therefore all interaction edges between them can be reflected locally. In addition we will require that in order for a file F to start being used for update by process P it is necessary that the subgraph $SG(F)$ to be brought over to $\text{location}(P)$ *. As a consequence all interaction edges between processes and files can be reflected locally. Therefore, the only interaction edges which cross site boundaries are those which represent interprocess communication between remote processes.

* This is not a necessary requirement and all the solutions presented in this chapter work correctly without it.

Given that the global condensed history graph G_c is partitioned into local subgraphs G_{loc} at each site we need a protocol by which updates to the graph are appropriately carried out.

5.3 - Update of the Local Subgraphs.

We will describe in this section the algorithms used to update the local subgraphs to reflect the following three operations.

- a. addition of a new historical version
- b. addition of an interaction edge
- c. intentional or accidental loss of an historical version.

5.3.1 - Addition of A New Historical Version

Since we are assuming that the subgraph $SG(x)$ is totally contained in G_{loc} for the site location(x), the operation of adding a new historical version is purely local to G_{loc} . This operation was described in section 4.6.1 of the previous chapter.

5.3.2 - Addition of An Interaction Edge

Here we have to distinguish between two cases. In one of them the interaction edge occurs between two nodes of the same subgraph Gloc. In this case the operation is strictly local and is described in section 4.6.2 of the previous chapter. We will restrict in this case the Cycle_Elimination routine to the local subgraph Gloc. While cycles which cross site boundaries may go undetected, the cost of detecting them every time that new potential cycles are generated may be fairly large. Alternatively one could "glue" all the subgraphs at a single site periodically (e.g. once a day, once a month or once a year) and detect all the inter site cycles discarding therefore the additional useless historical versions.

The second case is the one in which the interaction edge occurs between two objects located at different sites. In this case we are not going to collapse the two nodes involved in the interaction edge right away but we will batch these operations. Collapsing two remote nodes requires at least one message in addition to the one associated with the interaction edge itself. Therefore collapsing nodes every time an interaction edge occurs at least doubles the traffic in the network. The collapsing operation will be carried out with the aid of an algorithm which will execute periodically in the background without interfering with the normal

computation.

Let x and y be two objects located at different sites and let xi and yj be the nodes associated with x and y between which the interaction edge took place. We will denote an interaction edge as a 3-tuple (xi, yj, t) where t is a timestamp which indicates the time at which the interaction between x and y occurred. Note that there is always a generator of an interaction edge. For instance, if the interaction edge represents an IPC then the sender process is the generator of the interaction edge. Similarly if a process performs an operation on a file then the process is the generator of the interaction edge. Note also that at site $\text{location}(x)$ the identity of the node associated with object y , namely yj , in the interaction edge (xi, yj, t) is not known. Given these considerations the algorithm used to add the interaction edge (xi, yj, t) is simply the following.

S1 - Add to Gloc at $\text{location}(x)$ an undirected branch t :
 $xi \leftrightarrow y$. Notice that since yj is not known to the generator of the interaction edge the name of the object, y , is used to label one of the endpoints of the added branch. The node y will be called a non-local node to $\text{Gloc}(\text{location}(x))$.

S2 - Add to Gloc at location(y) an undirected branch t: $x \leftrightarrow y_j$. In general, it is not known at location(y) the identity of node x_i . The node x is a non local node to $Gloc(location(y))$.

The above operations are illustrated in figure 5.1. During the collapse operation, to be described next, nodes x_i and y_j will be collapsed into a super node labeled $\{x_i, y_j\}$. Note that the timestamp t along with the pair (x, y) serves to indicate which nodes, namely x_i and y_j , should be collapsed.

Since a fairly large number of interaction edges are typically generated per unit of time we do not want to keep the super nodes split apart. Therefore, periodically a protocol, called the Collapsing Protocol, will be executed to collapse nodes.

5.3.2.1 - Collapsing Protocol

Before we describe the Collapsing Protocol we must introduce the notions of sublog and global log and also describe the messages involved in the protocol.

Every site, S_i , keeps a list or sublog, $SL(S_i)$, of interaction edges which occurred after the previous collapsing operation took place. Let Tprevious be such a time. An entry in this sublog is of the form:

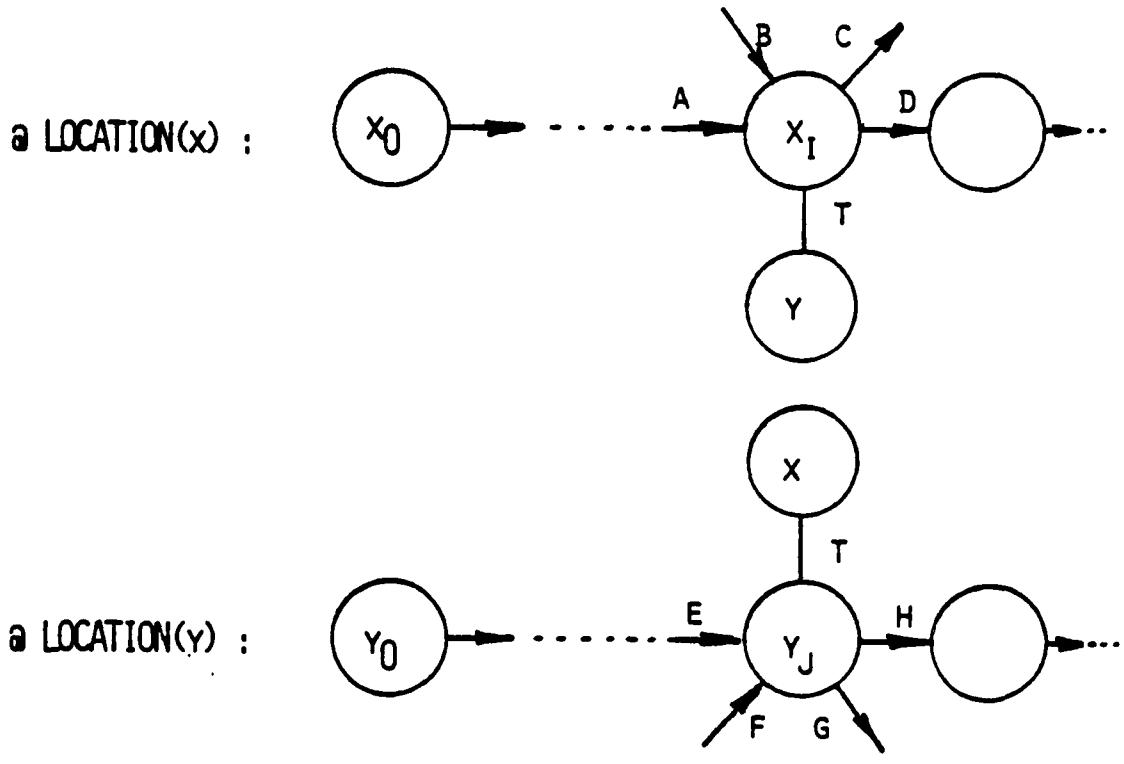


Figure 5.1 - Adding the Interaction Edge (x,y,t).

```
{ (xi,y,t), into(xi), out(xi) }
```

where xi is a node in the subgraph at site Si , (xi,y,t) is an interaction edge and $into(xi)$ and $out(xi)$ are as defined in the previous chapter.

The collapsing operation consists of two parts. During the first one, a global log, $GL(T)$, of all the interaction edges which occurred after $T_{previous}$ and no later than a certain time T is constructed. The set of all the nodes that should be collapsed as a result of the interaction edges represented in the global log is then calculated and the relevant subgraphs are updated in a subsequent part.

The algorithm used to calculate the set of collapsed nodes and their sets of input and output branches is straightforward. The input to this algorithm is a global log $GL(T)$ given as a sequence of interaction edges of the form (xi,y,t) as defined previously. Also, the global log, which is the union of sublogs, contains the into and out sets for all the nodes which appear as end nodes of interaction edges in $GL(T)$.

Even though a tuple such as (xi, y, t) does not specify completely an interaction edge the global log has the property that if there is a tuple (xi, y, t) in $GL(T)$ then there must also be a tuple (x, yj, t) in $GL(T)$. This implies that the interaction edge occurred between nodes xi

and y_j . These nodes should therefore be collapsed into a single node.

The procedure to find the set of all the collapsed nodes consists basically of first matching all the interaction edges in $GL(T)$. Then a super node is formed by collapsing a maximal set of nodes connected by interaction edges in the global log. The label of a super node is the list of the names of all its component nodes. The set of branches incident into a super node z is the union of the sets of branches which are incident into the component nodes of z from nodes other than these. Similarly, the set of branches incident out of z is the union of the sets of branches which are incident out of the component nodes of z but which are not incident into these nodes. Self-loops and parallel branches should be eliminated here.

The following example illustrates the above described procedure. Consider three objects a , b and c and consider the global log shown below.

interaction edges: $(a_1, b_1, t_1), (a^*, c, t_3), (b_1, a, t_1),$
 $(b_1, c, t_2), (c_1, b, t_2), (c^*, a, t_3)$.

adjacency sets:

```
into(a1) = {a0 -> a1}; out(a1) = {a1 -> a*}  
into(a*) = {a1 -> a*}; out(a*) = 0  
into(b1) = {b0 -> b1}; out(b1) = {b1 -> b*}
```

```
into(c1) = {c0 -> c1}; out(c1) = {c1 -> c*}
into(c*) = {c1 -> c*}; out(c*) = 0
```

Figure 5.2 shows the two super nodes obtained from the global log above. Figure 5.2.a shows the construction of the super nodes, figure 5.2.b shows the construction of the into and out sets for each of them and finally figure 5.2.c shows the two super nodes with their into and out sets and with parallel branches already eliminated.

The collapsing protocol uses two types of messages, called BUILD_LOG or BLOG and COLLAPSE_REQUEST or CREQ. The BLOG message is used during the first part of the protocol to collect all the entries in the global log GL(T). This message has four parameters as defined below.

S_i is the identification of the site which generated the BLOG message,

T is the time instant up to which new interaction edges are going to be considered for the global log to be constructed (this time instant will be called the time limit),

CL or Circulation List is the list of sites to which the message must be sent. The BLOG message will visit the sites in CL sequentially. This list will be updated as the message passes through the several sites as will be seen later.

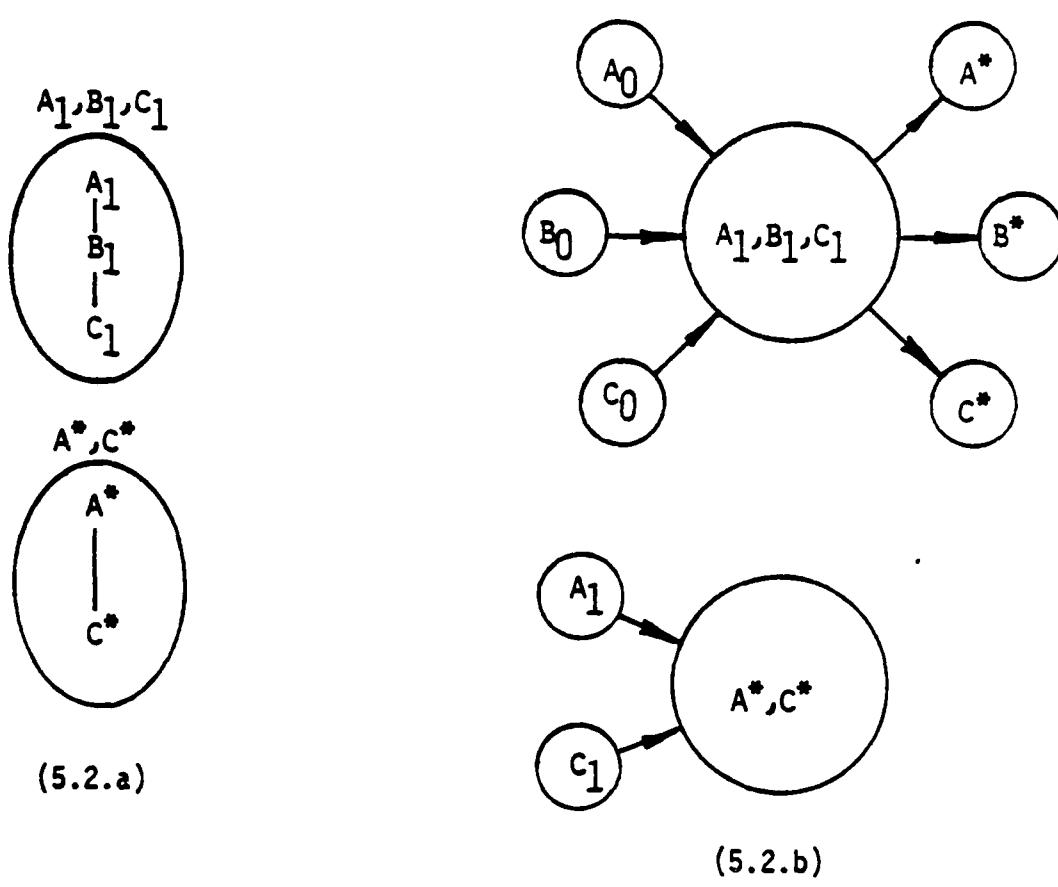


Figure 5.2 - Example of the Node Collapsing Operation.

PGL is a Partial Global Log. When all the sites in the Circulation List have already been visited and the list cannot be updated anymore then PGL is equal to the desired global log GL(T).

The pair (Si,T) serves a unique netwide identifier for the BLOG message.

The CREQ message is used in the second part of the protocol to send the set of collapsed nodes which resulted from GL(T) to all the sites which contributed to the construction of the global log. This message has the four parameters defined below.

Si is the identification of the site which generated the CREQ message,

T and CL are the same as defined for the BLOG message and

SSN or Set of Super Nodes is the set of super nodes obtained as a result of collapsing nodes according to the interaction edges in GL(T).

At any site there may be several messages (of either kind) at any given time. Messages are assigned one of the following four statuses:

1. OAS: Outstanding and Already Sent.

2. SAS: Saved and Already Sent.
3. STBS: Saved To Be Sent.
4. OPH1: Outstanding PHase 1.

Each site S_i maintains a queue, $Q(S_i)$, of saved messages, i.e. messages which have either status SAS or STBS. At each site S_i there is at most one outstanding message, denoted $M_{outs}(S_i)$, which has either status OAS or OPH1. The assignment of statuses to messages will become clear as we describe the protocol.

5.3.2.2 - Collapsing Protocol - Intuitive Description

The collapsing protocol is composed of two parts: a global log construction part and a graph update part. The global log construction part can be started by any site. The starting site will select a time limit T and will generate a BUILD_LOG message which will travel sequentially through all the sites which contain the subgraphs associated with objects which interacted since $T_{previous}$ up to time T . During its course the BLOG message will gather the sublogs accumulated at each relevant site in order to produce the global log.

Since another site may have started the same process with a possibly different time limit T' we have a race con-

dition which is resolved by giving priority to the message with the smallest time limit. Notice that no synchronization messages are necessary to deal with this race condition.

Consider two messages M1 and M2 with time limits T1 and T2 respectively. Let $T_1 < T_2$. If M1 arrives at a given site and finds that M2 is outstanding with status OAS then M2 will be saved with status SAS and M1 will be allowed to proceed becoming outstanding at that site. If M2 arrives at a given site and finds M1 as outstanding message then M2 is saved with status STBS.

Once all the sites have contributed their share to the global log, the last site to make a contribution will have the complete log. It then determines how nodes should be collapsed according to the interaction edges indicated by the global log. The result of this computation is essentially a set of updates that should be done in all relevant subgraphs. These updates are carried out through the use of a "nested two-phase commit" protocol [GRAY 78] since we want to make sure that either all the subgraphs are updated or none of them is. In other words we want this update to be an atomic operation. This atomic update is accomplished by the CREQ message which will circulate through the set of sites which participated in forming the global log.

When the subgraph at site S_i is updated to reflect all the interaction edges up to time T , the queue of saved messages at this site is examined. If the queue is not empty we select the message with the smallest time limit and make it the outstanding message. If its status is SAS its status is then changed to OAS. If on the other hand its status is STBS we must allow the message to continue its way by sending it to the next site to be visited. But before the message continues one must delete from its partial global log all the entries which have a timestamp not greater than $T_{previous}$ which is now updated to the time T .

In the following subsection we give an algorithmic description of the protocol.

5.3.2.3 - Collapsing Protocol - Algorithmic Description

The collapsing protocol as executed at site S_i is described by the rules given below. It is assumed that there is a virtual circular order of the sites such that when a message in this protocol has to be sent to a list of sites, it is sent to the site in the list which is next in the circular order.

Rule 1: Site Si decides to start a collapsing operation. There can be no outstanding messages at site Si. Let Tclock be the value of the clock.

S1.1 - Select a time limit $T > T_{previous}$ (typically T will be selected to be equal to Tclock).

S1.2 - Build a BLOG message M with the partial global log PGL(M) initialized to the sublog SL(Si) and with the circulation list CL(M) initialized to the list of sites which contain the subgraph SG(x) if x is an object involved in SL(Si). Mark site Si as "visited" in CL(M).

S1.3 - Send M to the next non-visited site in CL(M).

S1.4 - Make M the new outstanding message at site Si and assign to it status equal to OAS (Outstanding and Already Sent).

Rule 2: A BLOG message $M = (Sj, T, CL, PGL)$ is received at site Si.

S2.1 - If there is no outstanding message at Si then go to step S2.4.

S2.2 - If the outstanding message at S_i has a time limit $T' < T$ then mark site S_i as "visited" in $CL(M)$, save the incoming message in the queue $Q(S_i)$ and assign to it the status STBS (Saved To Be Sent). Stop.

S2.3 - [at this step there is an outstanding message with status OAS and time limit $\geq T$] Save the outstanding message in $Q(S_i)$ and assign to it status SAS (Saved Already Sent). The incoming message M is now the outstanding message.

S2.4 - [process incoming message] Add the sublog $SL(S_i)$ to $PGL(M)$. Mark S_i as "visited" in $CL(M)$. Add to $CL(M)$ the set of all the additional sites which are involved because of interaction edges in the sublog $SL(S_i)$.

S2.5 - If there are non visited sites in $CL(M)$ then send the message M to the next site in $CL(M)$, make $status(M) = OAS$, and stop.

S2.6 - [all the sites in $CL(M)$ are visited: the partial global log, $PGL(M)$, is now the global log $GL(T)$] Find the set, SSN , of all the super nodes derived from the global log $GL(T)$. Discard message M .

S2.1 - [the "graph update" phase can now be started] Build a CREQ message $M' = (S_i, T, CL, SSN)$ where $CL(M') = CL(M)$ and the time limit T is the same as in the incoming message M . Mark all the sites in $CL(M')$ as "non-visited" excluding site S_i .

S2.2 - Send the CREQ message to the next site in $CL(M')$. The message M' is now the new outstanding message and is assigned status = OPH1 (Outstanding PHase 1).

Rule 3: A CREQ message $M = (S, T, CL, SSN)$ is received at site S_i .

S3.1 - If there is an outstanding message M' at site S_i which is not a CREQ message then make the incoming message M the new outstanding message with status equal to OPH1 and discard message M' else go to step S3.4 .

S3.2 - Write intentions list for updates to the local subgraph as indicated by $SSN(M)$. Set $T_{previous}$ to T .

S3.3 - Mark S_i as "visited" in $CL(M)$ and send M to the next non visited site in $CL(M)$ if there is any otherwise send it to site S . Stop.

S3.4 - [the outstanding message is a CREQ message with $S = Si$: end of phase 1 of two-phase commit] If the outstanding message M' is equal to the incoming message then mark the sites in $CL(M)$ as "non-visited" except for Si .

S3.5 - [The outstanding message is a CREQ message with $S \neq Si$] Carry out local updates on local subgraph as indicated by the set SSN in the message M . Mark Si as "visited" in $CL(M)$.

S3.6 - [adjust sublog to account for update in local subgraph] Delete from sublog $SL(Si)$ all the entries with a timestamp $\leq T_{previous}$.

S3.7 - Forward M to the next non visited site in $CL(M)$ if there is any.

S3.8 - Discard M and select the message M'' in $Q(Si)$ with the smallest time limit (if any) to be the outstanding message.

S3.9 - If $status(M'') = SAS$ then change its status to OAS.

S3.10 - If $\text{status}(M'') = \text{STBS}$ then delete from $\text{PGL}(M'')$ all the entries with timestamp $\leq T_{\text{previous}}$, forward M'' to the next non visited site in $\text{CL}(M'')$ and change $\text{status}(M'')$ to OAS.

Figure 5.3 illustrates how the subgraphs of figure 5.1 look like after the collapsing protocol has been executed. All the nodes flagged with a star(*) are called non local nodes since they do not belong to the subgraph of any local object.

5.3.2.4 - Collapsing Protocol - An Assertion

Assertion: The collapsing protocol applies the updates generated by global logs in increasing chronological order of time limit.

Proof: We will say that a global log, $\text{GL}(T)$, completes when the BLOG message has already passed through all the relevant sites. In order to prove this assertion it is only necessary to prove that global logs complete in increasing chronological order. Assume not. Then there are two global logs $\text{GL}(T_1)$ and $\text{GL}(T_2)$ with $T_1 < T_2$ and such that $\text{GL}(T_2)$ completed before $\text{GL}(T_1)$ did. Since $T_2 > T_1$, $\text{GL}(T_1) \subseteq \text{GL}(T_2)$ and therefore, all the sites that must be visited by the BLOG message associated with $\text{GL}(T_1)$ must also be visited by the BLOG message associated to $\text{GL}(T_2)$. But since priority is given to messages of smaller time limit, $\text{BLOG}(T_1)$ could

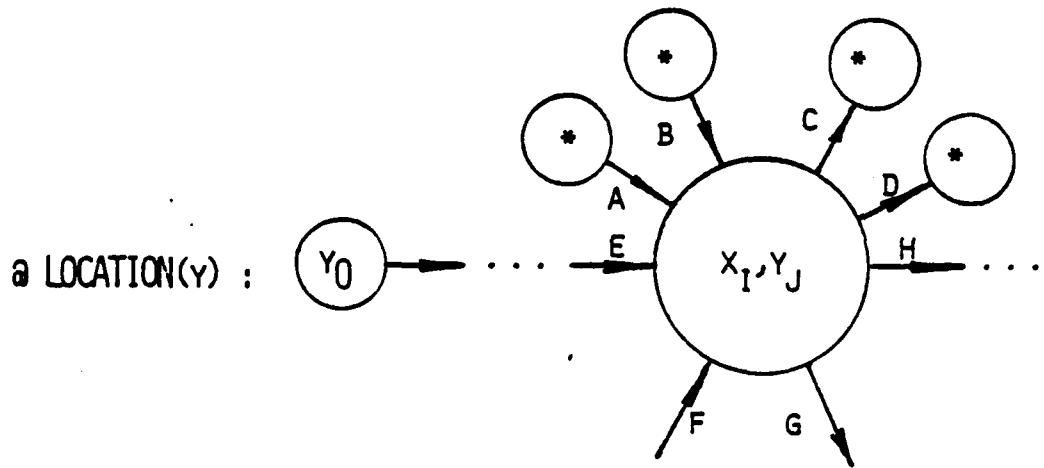
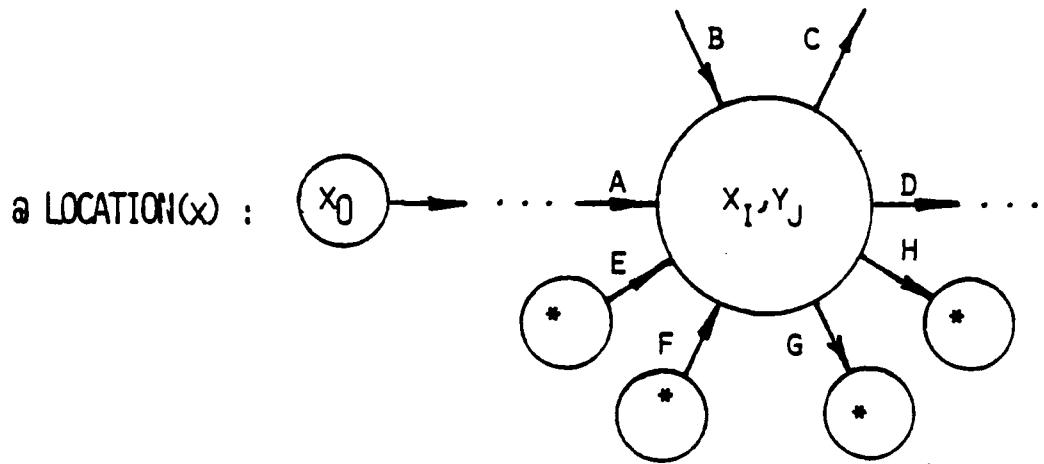


Figure 5.3 - Example of the Application of the Collapse Protocol to the Graph of Figure 5.1.

not have been generated before BLOG(T2) visitee all the sites which should be visited by BLOG(T1) otherwise BLOG(T2) could not complete before BLOG(T1). Also, BLOG(T1) cannot be generated by any site S_i before the updates due to GL(T2) are applied to the subgraph at site S_i (see Rule 1). But when these updates are applied, $T_{previous}$ is set to T_2 which is greater than T_1 . Since no BLOG messages with time limit $< T_{previous}$ can be generated after this point. Therefore, BLOG(T1) cannot be generated after this point. Hence, BLOG(T1) was never generated. This is a contradiction and proves the assertion.

5.3.3 - Intentional or Accidental Loss of An Historical Version

Discarding an historical version is equivalent to collapsing the two end nodes of the branch which represents the historical version in the graph. Due to the partitioning criterium used to decompose the global graph G_c such end nodes are both located in the same subgraph. Therefore, this operation is strictly local and was described in section 4.6.2 of the previous chapter. The same comment made in section 5.3.2 of this chapter regarding the Cycle_Elimination routine applies here as well.

5.4 - Distributed Crash Set Calculation

One of the major results of the previous chapter is that given a set of objects X , $\text{crash_set}(X)$ is given by the set of branches incident into nodes of the graph $G^*(X)$. The graph $G^*(X)$ was found by using a depth first search (DFS) graph traversal procedure. In our case, in which the global graph is distributed, we have to find a distributed algorithm to find the graph $G^*(X)$ and consequently $\text{crash_set}(X)$. This algorithm can be intuitively described as follows.

Let S_i be a site where an error was detected. Let X be the set of objects detected to be in error. The local subgraph at S_i , $G_{\text{loc}}(S_i)$, will be traversed in a way similar to the one described in the previous chapter. The difference now is that in each local subgraph there are two types of nodes, namely local and non-local ones.

The DFS procedure will be executed at site S_i and each time that a non-local node is visited it will be marked. These marked nodes are called output connections and may be used as starting nodes or roots for the execution of a DFS procedure at a foreign site.

Once all the possible nodes in $G_{\text{loc}}(S_i)$ have been visited we have a collection of output connections and a set of sites where these output connections are local nodes. At each of these sites, the DFS procedure is invoked for each

of the roots provided that they have not been already visited in any of the previous invocations of the DFS procedure at that site. Again if non-local nodes are marked the process repeats itself. If however, only local nodes are visited at a given site by the execution of the DFS procedure then this site returns to its "calling site" the set of nodes visited locally as well as those visited at sites called by this site. By "calling site" we mean the one which triggered the execution of the DFS procedure at a given site. Eventually, the first site to start the crash set calculation will receive from the sites which were initially triggered by it sets of nodes the union of which is the node set of $G^*(X)$.

The above algorithm will be best understood in the light of an example. Consider the global graph G_c of figure 5.4. Graphic conventions were used to denote how G_c is partitioned among three sites S_1 , S_2 and S_3 . Nodes of G_c are labeled with node numbers which are enclosed in brackets. Let us assume that we want to find $\text{crash_set}(X)$ where X contains a single object associated with node 1 at site S_1 . The progress of the algorithm can be graphically summarized by a tree, called the Progress Summary Tree (PST), in which each node is associated with a possible root for a DFS procedure invocation. The sons of a node x in the tree are the output connections found when applying DFS starting at node x . A node x in this tree is labeled by the set of nodes

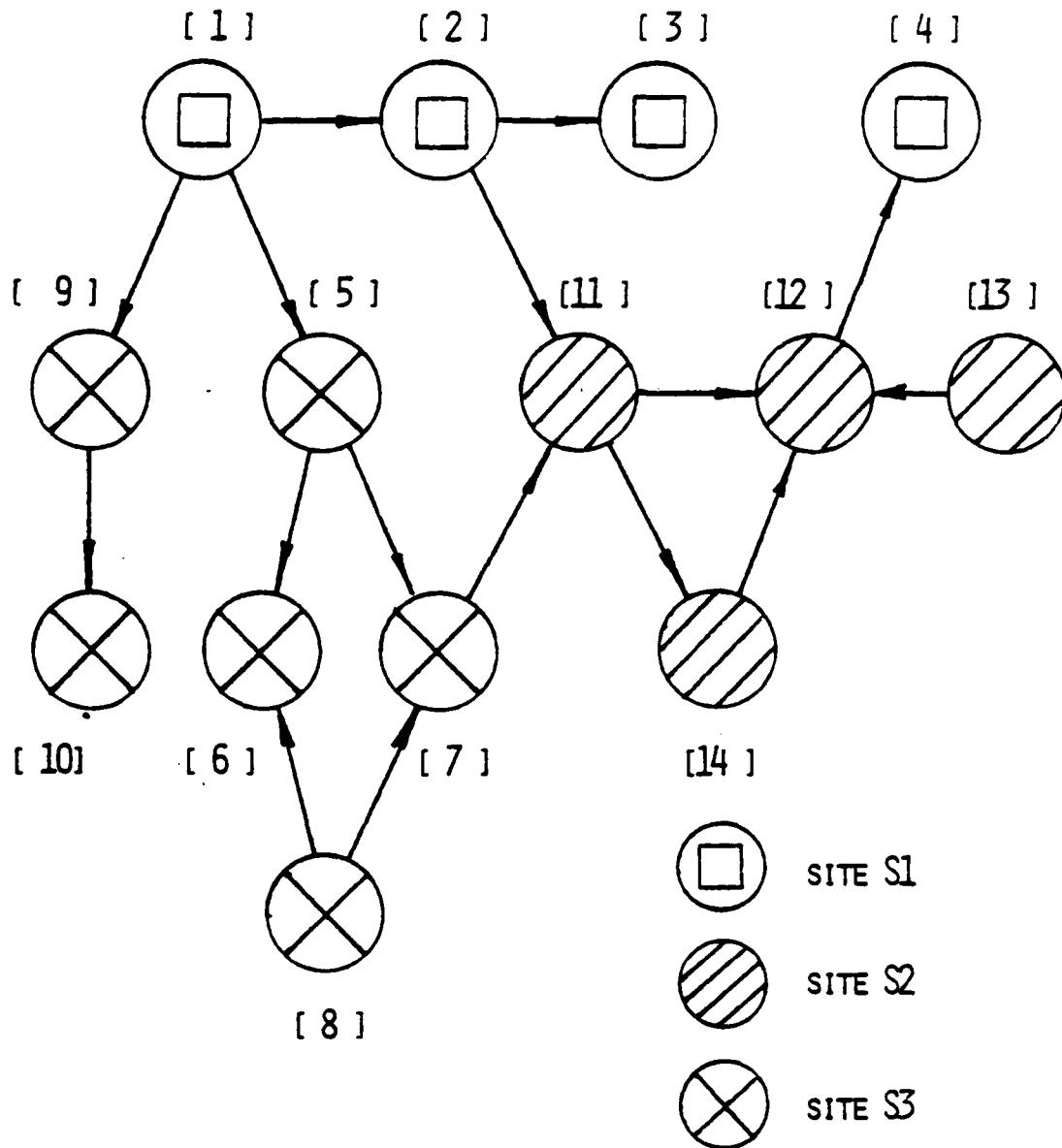


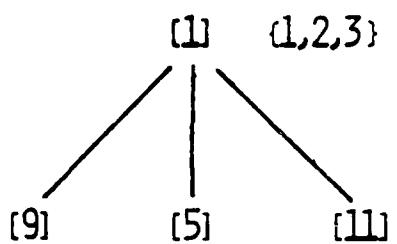
Figure 5.4 - Global Graph G_c Distributed Among Sites S₁, S₂ and S₃.

visited by the DFS procedure if x was used as a root.

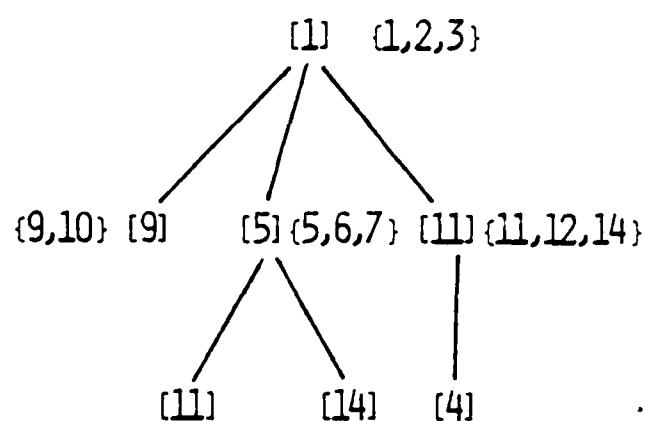
In the example we are considering, DFS is applied at site S1 starting at node 1. Three output connections are found, namely node 11 at site S3 and nodes 5 and 9 at site S3. The PST at this moment is shown in figure 5.5.a .

A message is sent to site S2 containing the root 11 and another message is sent to site S2 containing the roots 5 and 9. At site S3, the DFS procedure is applied twice: once starting at node 9 and another starting at node 5. Two output connections are found, namely nodes 11 and 14 at site S2. A message containing these nodes is sent to site S2. Notice that node 11 had already been marked as an output connection at site S1. Since nodes already visited by the execution of the DFS procedure are not used as roots, node 11 will be used as a root only once. Let us assume in this case that node 11 was used as a root in site S2 as a result of the message from S1. An output connection to node 4 is found and a message is sent to site S1. The PST up to this point is given in figure 5.5.b .

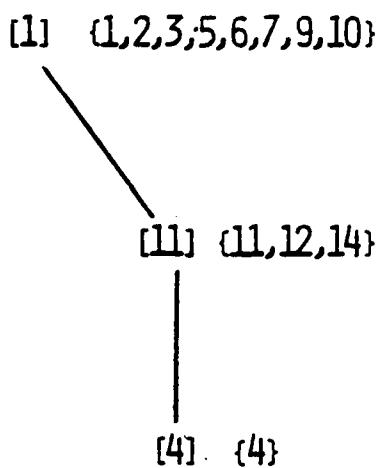
Since nodes 11 and 14 were visited while executing the DFS procedure at site S2 upon request of site S1, the request of S3 to S2 has no effect and this completes all the processing that could possibly be carried out at S2 or at sites invoked by it. Therefore, S2 sends a message to S1 with the set of nodes visited at S2 and at the sites invoked



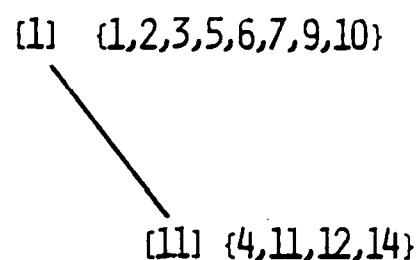
(5.5.A)



(5.5.B)



(5.5.C)



(5.5.D)

[1] {1,2,3,4,5,6,9,10,11,12,14}

(5.5.E)

Figure 5.5 - PSTs for the Application of the Distributed Crash Set Calculation Algorithm to the Graph of Figure 5.4.

by S2. This set is {5, 6, 7, 9, 10}. At site S1, the application of the DFS procedure yields no output connection. Figure 5.5.c depicts the progress summary at this point.

Now, site S1 returns to site S2 (see figure 5.5.d) and finally site S2 returns to site S1 (see figure 5.5.e). The label of node 1 in figure 5.5.e is the set of nodes of $G^*(X)$.

The algorithm just described exhibits a fairly large degree of parallelism since it performs all possible local computation and then triggers foreign computation at several sites to explore all the possible concurrency inherent to the problem. The overall graph traversal procedure can be viewed as a mixture of depth first search for intra site graph traversal and branch and bound for inter site graph traversal.

An Algol-like description of the algorithm is given in Appendix B.

5.5 - Conclusion

This chapter addressed the problem of finding the crash set of a set of objects in distributed environments. Since in distributed systems we cannot assume that there is a single and complete copy of the condensed history graph G_c , a partitioning criterium is given whereby each site main-

tains a subgraph of G_c . A protocol to update these subgraphs when interactions edges are generated was given here. Also, a distributed algorithm to find the crash set was intuitively described in this chapter, while a complete description of the algorithm can be found in Appendix B of this dissertation. The protocol explores all the possible parallelism which exists due to the fact that the graph G_c is partitioned and that there are several sites which may be concurrently doing their part in the crash set calculation.

CHAPTER 6
THE UCLA DISTRIBUTED SECURE SYSTEM BASE

6.1 - Introduction

In this chapter we will show how the techniques and models for crash recovery discussed in the two previous chapters, were used in the design of the UCLA Distributed Secure System Base (DSSB). The UCLA DSSB, which is described in detail in [MENA 78c], is intended as a base for secure and reliable distributed computing. The belief is that applications such as office automation systems, distributed database management systems and the like can be built more easily on top of a base such as the DSSB.

The system can be thought as a collection of loosely coupled homogeneous processors. Running at each processor there is a slightly modified version of the UCLA Data Secure Unix Security Kernel [KAMP 77], which is a portion of the UCLA Data Secure Unix operating system. The kernel contains all the mechanisms necessary to enforce authorized access to the objects in the system (e.g., core frames, pages of files, processes, message channels, etc.).

Reliable computation is achieved by the provision of automatic backup of processes and/or files to previously saved snapshots, when errors are detected. In the following sections we will discuss the architectural principles used

in the design of the DSSB and then we will describe the actual system design. This description emphasizes the modules of system which are responsible for crash recovery and concurrency control since they constitute the major contribution of this author to the system design. Nevertheless, enough detail of the remaining modules is given in this chapter to help to set the context appropriately. A detailed description of the File Policy Manager - one of the system modules - can be found in [RUDI 79].

6.2 - Architectural Principles

This section describes the principles upon which the architecture of the DSSB is built.

6.2.1 - Network Independence

In order to recover from crashes in distributed systems one may want to restart a process at a foreign site if the site where it was running crashes and if an historical version of the process is available for this purpose. Therefore, for ease of recovery, processes at any site should be allowed to refer to any object by its virtual name, i.e. a name which contains no information about the network. This principle of network independence suggests a network wide name space. It, together with high bandwidth communication lines, suggests a network wide page faulting mechanism.

6.2.2 - Multiple Copies

Crash recovery in a distributed system requires that copies of objects be stored at more than one site in order to allow for continued operation in the face of site and communication link outages. The existence of multiple copies of objects calls for a mechanism or protocol for the management of those copies in a distributed environment. This protocol should be implemented at the lowest possible level within the Recovery Software so that most of it and also all of the user software is able to refer to virtual objects only, without any concern for the existence of multiple physical realizations of them.

A protocol to manage multiple copies of objects should address the following issues:

a. Locking for synchronization of access. A set of rules through which processes can acquire and release control over an object must be specified. The basic alternative solutions to this problem are:

a.1 Centralized Control. Access requests are sent to a centralized controller for the object which is responsible for making decisions as to whether access to the object should be

granted or denied. This approach is clearly unacceptable for a distributed system since a crash of the centralized controller would bring the whole system to halt. This problem is solved by the next alternative.

a.2 Adaptive Centralized Control. In the face of a crash (actual or apparent) of the centralized controller a new one should take its place. Therefore we need a mechanism for the election of a new controller. Moreover, due to network partitioning one may end up having one controller per partition, which requires the existence of a mechanism to merge partitions upon network reconnection. The structure of such a protocol would almost certainly be too complex for the purpose of locking multiple copies of objects in a distributed system. An example of an adaptive centralized protocol for the more general case of locking in distributed databases was presented in section 3.2 of chapter 3.

a.3 Distributed Control. The set of sites which store a copy of the object in question should agree globally before any one of them can be given control over the object. This approach

appears to have the potential of overcoming the difficulties mentioned above, if a simple protocol with suitable performance characteristics can be developed.

b. Rules for operation in the face of network partitioning. These are the rules which specify when operations should be allowed to proceed in the event that some of the participating sites cannot be reached. Those rules are extremely important in the case of actual or apparent network partitioning. The two basic alternatives in this case are:

b.1 Allow each partition to proceed independently of the others and try to merge operations (if possible) when the network is reconnected. This solution may require that some fraction of the amount of operations performed while the network was partitioned have to be backed up.

b.2 Use some rule which uniquely determines which partition should be allowed to proceed. As an example one could specify that only the partition containing the majority of sites should be allowed to proceed. To allow maximum use

of the system in the face of partitions, the majority protocol could be applied on a per object basis.

c. Mutual Consistency. A set of copies of an object is said to be mutually consistent if all of them converge to the same value after all activity ceases. Alternatively, one could guarantee that all of the sites know the name and location of the most current copy of the object. Therefore, each site should be able to detect the fact that it has an outdated copy of the object and request of the appropriate site that the current version be made available.

The protocol to be utilized in our architecture has distributed control, uses a weighted majority rule for continued operation in the face of network partitioning and handles mutual consistency by guaranteeing that every site is able to know the name and location of the most recent copy of a given object.

6.2.3 - Error Confinement

Systems designed with reliability in mind should provide for fire walls against widespread damages. In other words, errors when they occur should be confined to the

least possible number of objects. Randell introduced, in [RAND 75], a device called a conversation, which is a generalization of the notion of a transaction in database systems [ESWA 76, LAMP 76]. Any set of objects which want to interact should agree to enter into a conversation. Objects within a conversation may interact freely, but may not interact with any other objects outside it. If a failure is detected in any object of a conversation, the whole conversation would be backed up to a recovery point taken at the entry point of the conversation. In this sense, a conversation is an atomic action. This implies that the outcome of the computation carried out by processes inside a conversation cannot be seen by processes outside it otherwise the conversation would not be atomic anymore. In other words, there cannot exist any information flow across conversation boundaries.

There are clearly two advantages in having conversations, namely:

1. they provide a mechanism for error confinement - a necessary requirement for any reliable system architecture.
2. they provide a mechanism for error recovery - i.e. they are the units of recovery.

Since one of the requirements of the DSSB is security, information flow can only occur between two objects if the protection policy allows the flow to occur. An information flow model like Denning's lattice model [DENN 76] or a model based on colors and profiles [POPE 77] is necessary to establish secure information flow if processes cannot be trusted. Notice that the protection mechanism establishes the necessary fire walls to avoid widespread damages caused by errors. If the color model is used one can change the users profile to limit his access rights temporarily. This would also limit the set of objects that would potentially be affected by a failure. It is important to note also that even if security were not an issue we would still need a mechanism to enforce the property that information flow do not cross conversation boundaries.

Note that requiring that processes outside a conversation do not see the result of the computation of a conversation until it completes successfully is likely to pose a serious performance degradation both during normal operation - through decreased concurrency- and during crash recovery - due to the necessity of complete backup of possibly long computations. In short, for medium and long duration conversations it is not efficient to commit the results of a computation only after it has successfully completed. This is an acceptable solution for the case of database management systems where the transactions are short (most of them

at least).

The crash recovery strategy implemented in the distributed system is based in backward error recovery. This implies that results are committed when generated and if errors are detected one has to find out the set of processes which used these results, back them up along with the results they produced and so on and so forth. The net result of using this strategy is that we end up having a finer grain of recoverability.

Since the issue of error confinement is taken care by implementing secure information flow mechanisms and since we do not want to have a conversation as an atomic action because of efficiency considerations it is not necessary to implement conversations in the DSSB.

6.2.4 - Minimization of Recovery Relevant Data

The model for recovery presented in chapter four of this dissertation indicates that the recovery data is composed of historical versions of objects and of interaction edges among those objects. One would like to be able to store the recovery data in files, but the files themselves should be restorable by the use of the recovery data! Thus, we have a circularity problem which is dealt with by making the following observations. Interaction edges are critical in the sense that if any of them are lost it is possible

that the calculation of a crash set generates a globally inconsistent state, since one or more interactions between objects could have been ignored.

On the other hand, the only problem associated with the loss of one or more historical versions for an object is a possible performance degradation since more backing up may be necessary to recover from a crash. Therefore, historical versions of objects will be stored and retrieved as ordinary files supported by the File Policy Manager (FPM), to be described later in this chapter.

Note that interaction edges do not need to be recorded as such once their effect in the global condensed graph has already been taken into account. This fact along with the fact that historical versions themselves may be lost without compromising correct error recovery, shows that the portion of the recovery data which is critical for correct error recovery is contained in the condensed history graph. A mechanism for reliably storing and updating this graph, using a stable storage notion [LAMP 76] is implemented at each site.

6.2.5 - Separation of Security Relevant from Recovery Relevant Mappings.

The complete mapping which indicates for each object which sites have a copy of the object and what is the actual

physical location of the object within each site is decomposed into security relevant local mappings and non-security relevant mappings to other sites. The former is a local mapping which indicates the actual physical location of each local object only. The latter indicates to each site what the other sites are which have a copy of the object, giving no indication of the actual physical location at those sites. Having each site know about the actual physical mappings at each other site would involve a considerable performance overhead for updating this information.

Figure 6.1 illustrates the decomposition mentioned above. In particular, figure 6.1.a shows the complete mapping for an object x , for which there are copies at sites S_1, S_2, \dots, S_k . Figure 6.1.b shows the decomposed mapping.

6.2.6 - Separation of Mechanisms from Policy

One would like to be able to verify the properties of consistency and correct recovery for the Recovery Software and for this purpose the architecture of the DSSB minimizes the dependency of the recovery process on the rest of the software. As an application of this principle, this design entirely separates recovery policies from recovery mechanisms. Recovery policy issues such as frequency of snapshots, number of copies to be kept of any object, and the like can be dealt with in as sophisticated a manner as

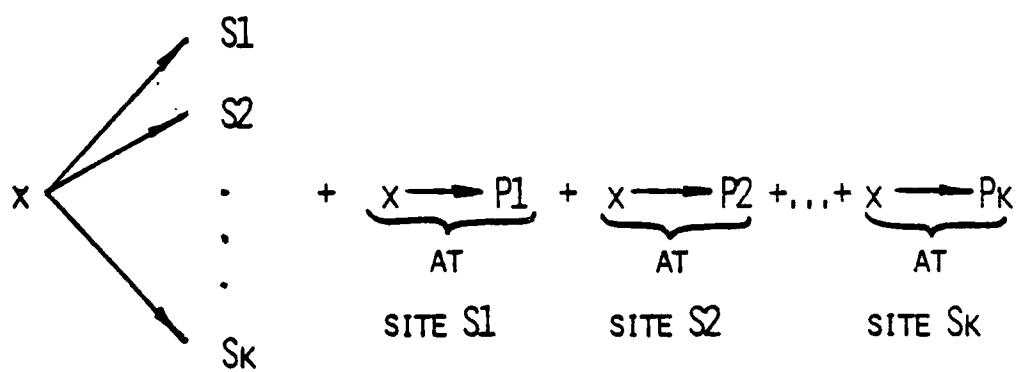
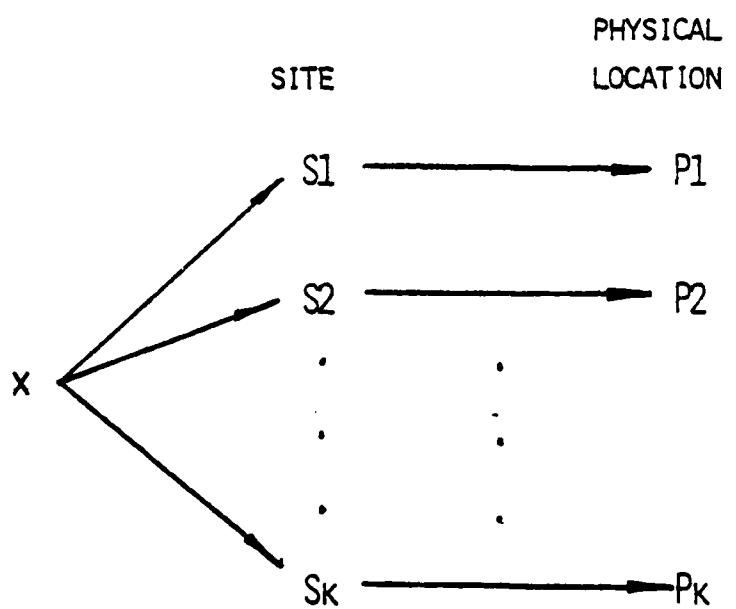


Figure 6.1 - Complete Object-Location Mapping (fig. 6.1.a) and Decomposed Mapping (fig. 6.1.b).

the user desires. These policies are implemented by modules which do not belong to the Recovery Manager, the part of the recovery software responsible for carrying out a correct crash recovery operation. The correct operation of modules that supply data used by the Recovery Manager is similarly not affected by policy software.

6.3 - Major System Modules

The Distributed Secure System Base is composed of four major modules:

- a. the Base Kernel (BK).
- b. the File Policy Manager (FPM).
- c. the Multiple Copy Manager (MCM).
- d. the Recovery Manager (RM).

The Base Kernel is a slightly modified version of the UCLA Data Secure Unix Kernel [KAMP 77]. It can be viewed as providing an abstract machine composed of several types of objects: processes, units of storage called segments, devices, message channels ; and operations valid on these objects, which are subject to user settable protection controls.

The File Policy Manager (FPM) implements security policy controls with the use of a secure information flow model. In addition to that, the FPM along with the Multiple Copy Manager and the Base Kernel implements a distributed file system. Interaction edges which involve files (i.e. openings and closing of files) are sensed by the FPM and reported to the Recovery Manager.

The Multiple Copy Manager (MCM) basically implements the majority consensus protocol, mentioned before, used to coordinate access to the multiple existing copies of files. The Recovery Manager (RM) essentially implements the crash recovery protocols introduced in Chapter 5.

Figure 6.2 illustrates the layered architecture of the DSSB. The understanding of the layering is that a module at any given layer only needs the operations provided by the layers at inner levels for it to work correctly. Each of the component modules of the DSSB will be discussed in the remainder of this chapter.

6.3.1 - Base Kernel (BK)

We will only discuss here the changes to the Base Kernel which were introduced to support the distributed system. These modifications fall into three categories:

- a. added kernel calls.

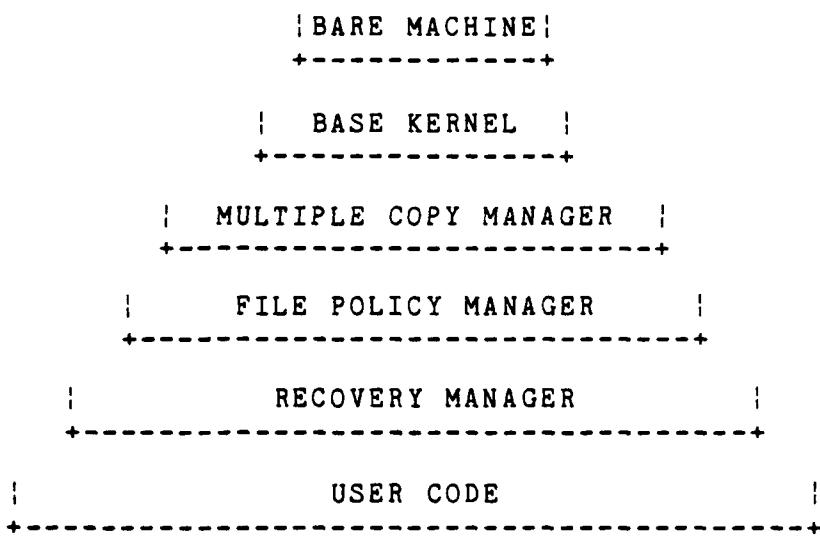


Figure 6.2 - Layered Architecture of the DSSB.

- b. modifications of capability representation.
- c. network wide inter process communication.

6.3.1.1 - Added Kernel Calls

Two kernel calls were added to the UCLA Data Secure UNIX Security Kernel in order to support the implementation of a network wide page faulting mechanism. These calls, which are described in detail in [MENA 78c], are the SHIP IN and SHIP OUT calls.

The SHIP IN call causes a page which is stored at a remote site to be brought into local main memory. This call is issued by the File Policy Manager, which allocates secondary storage space for the requested page. A page frame designated to hold the incoming page must also be made

available before the desired page is requested to the foreign Base Kernel. Remote kernels refer to the page by its virtual name which is net wide unique and site independent.

When a kernel receives a SHIP IN request from a foreign kernel it checks first whether the requested virtual page is core resident. If this is the case, the kernel starts transferring the page to the requesting site. If not, the SHIP IN request is passed to the FPM which finds the disk location of the page and issues a SHIP OUT call

Upon receipt of the SHIP OUT call, the kernel causes the page to be brought to core and transfers the page to the destination (requesting) site.

6.3.1.2 - Capability Representation

The architecture of the Base Kernel is capability based. In order for backward error recovery to take place conveniently, it is necessary that processes be able to migrate and still refer to the same objects in a site independent manner. Therefore, the name field in the capability has to be a virtual name. Other fields in the capability are:

- a. local disk location: for pages only.

- b. core location guess pointer: for pages only - this field is a guess as to where in core the page can be found.
- c. access rights.

6.3.1.3 - Network Wide Inter Process Communication

The Base Kernel due to its functions and being at the lowest level in the system hierarchy must be aware of the existence of the network and must know how to communicate with objects at remote sites by using the network explicitly. As a consequence, the set of Base Kernels are in a good position to make the network transparent as far as processes are concerned. In other words, the set of BKs implement a network wide inter process communication mechanism which along with a network independent naming scheme for processes makes inter process communication the same, be it local or remote.

6.3.2 - Multiple Copy Manager (MCM)

The MCM is primarily responsible for implementing a protocol which can be used to coordinate access to multiple copies of files. An intuitive description of the majority consensus protocol is given in the remainder of this subsection.

The multiple copy management protocol implemented by the MCM integrates all the aspects of the problem discussed in section 6.2.2 of this chapter. Control is distributed in the sense that the majority of sites which keep a copy of a given file has to be involved in the process. We will assume throughout this discussion that the set of sites that maintain a copy of a given file is static. More will be said about this later. We will also assume the existence at each site of a process called a Multiple Copy Manager (MCM) which actually implements this protocol. This protocol is homogeneous in the sense that it is the same for all nodes. The approach taken here to mutual consistency is that of making sure that every MCM is able to know who has got control over a file, what is its status and what is the name of its most current copy.

Each site keeps a data structure called a status descriptor (SD) which indicates the status of the system as viewed by that site. The status indicates whether the file is locked or not and in the affirmative case, in which mode it is locked, and also the name of its most current copy. An SD is a 4-tuple of the form $\langle ts, fn, mode, tout \rangle$ where,

1. ts is a timestamp whose use will become clear as the protocol is described.
2. fn is the file name which is a two part name of

the form <global name>.<generation#>.

3. mode is the mode in which the file is locked. It can assume one of the four possible values: unlocked (U), share (SH) and exclusive (X).
4. tout is an estimate of the time interval during which the file will be in use. When mode = U then tout is set to -1.

A status descriptor with mode = U will be called an unlocked status descriptor as opposed to a locked SD for the case in which mode \neq U.

For the purposes of this explanation it is necessary to consider only one file. Let it be called F and let S = {S1, S2, ..., Sn} be the set of sites that keep a copy of F. Assume that the initial state of the system is such that all SDs have the following value <t0, F.0, U, -1> indicating that a copy of F with generation number equal to zero was created at all sites at time t0 and that no one has control over F.

Let S1 be a site who wants to lock file F. In order to determine the status of the file it requests from all sites in S that their current SD for F be sent to S1. After S1 has collected SDs from at least the majority of sites in S (including itself) it selects the most current status

descriptor, i.e., the one with the biggest timestamp. As will be proved later, the correct status of a file can be correctly determined by the most current SD among a set of SDs collected from a majority of sites. Let SDmax be this most current status descriptor.

If SDmax is an unlocked status descriptor or if it is locked in a non-conflicting mode, then S1 is allowed to lock F. It does so by successfully broadcasting to at least the majority of sites an appropriately timestamped SD which indicates that S1 is the locking site. An estimate of the time interval during which S1 will need F is included in the timeout component of this SD.

If SDmax is locked in a conflicting mode then S1 has to wait and retry later. The timeout component of SD1 can be used as a guess to when S1 should retry.

A site releases a lock on the file F by reliably broadcasting to at least the majority of sites a release request. A protocol similar to the Assured Communication Protocol (see section 3.2.2 of chapter 3) is used to provide the needed robustness to the release control protocol.

The period during which F will be used by S1 can be renewed by sending an appropriately timestamped renewal request and having it being approved by at least the majority of sites. If S1 fails to get a majority for its renewal re-

quest it must release control over the file. If the file was locked in exclusive mode by S1, then S1 must backup the updates it did on F. When a site times out for its current SD on F, it will discard it and replace it by the previous SD. This mechanism allows locks on files to be released from a set of sites which do not constitute a majority, thereby allowing the majority to use it.

Only one site should be able to lock a file in exclusive mode. In order to enforce this property it is sufficient that a MCM only sends an unlocked status descriptor as a reply for a lock request to one and only one MCM. If an unlocked SD had already been sent to another site when a request for locking the file in exclusive mode is received, a message is sent to the requesting site that someone is already trying to lock the file in exclusive mode. But now, we have a potential for deadlocks since several sites may collect less than a majority of unlocked SDs each. Therefore none of them would be granted exclusive access to the file and they would not be able to proceed. The solution to this problem is to have each site retry again after a randomly chosen delay.

As we asserted before, the examination of the most current status descriptor, i.e., the one with the biggest timestamp, determines correctly the status of a file. Let us now prove this assertion.

ASSERTION 6.1: Let $S = \{S_1, S_2, \dots, S_n\}$ be a set of sites at which there are status descriptors for a file F. The SD with the biggest timestamp selected from a subset X of S containing a majority of the sites in S reflects the current status of the file.

Proof: In the MCM protocol described above, every time that the status of a file F is changed (i.e. the file is locked or unlocked), a subset of S containing at least a majority of sites has to be notified. Let Y be the subset of S which was involved in the operation which resulted in the last change to the status of the file. Therefore, Y contains at least a majority of the sites in S and the following inequality holds:

$$|Y| \geq [|S|/2] + 1 \quad (6.1)$$

where $[x]$ is a function which gives the greatest integer less than x. The SDs in all sites on Y are the same and they reflect the current status of the file F. Since the set X also contains a majority of the sites in S the following inequality is true:

$$|X| \geq [|S|/2] + 1 . \quad (6.2)$$

While X and Y may be different, in general, they must have at least one element in common. Assume not. Then, it must be the case that

$$|X| + |Y| \leq |S| \quad (6.3)$$

But if we sum (6.1) with (6.2) we get

$$\begin{aligned} |X| + |Y| &\geq 2*[\lfloor |S|/2 \rfloor] + 2 \geq \\ &\geq |S| - 1 + 2 \geq |S| + 1 \end{aligned} \quad (6.4)$$

which contradicts (6.3) and proves that X and Y are not disjoint. The assertion is then proved since the SDs in Y have the biggest timestamp and therefore reflect the current status of the file F and since $X \cap Y \neq \emptyset$.

Some observations about the above described protocol are in order. First, it is not necessary for all the sites where a copy of the file exists to be involved in the decision procedure. In particular, one can designate a subset of the set of sites which have a copy of the file - let us call them voting sites - as being those which are involved in the protocol and with respect to which a majority is considered. This approach cuts down on the communications cost incurred by the protocol.

A second observation is that the majority rule solves the problem of network partitioning by allowing the partition with a majority of voting sites for a given file to continue using the file. This rule may be made more flexible by assigning weights to each voting site and by considering a weighted majority. As an example, one may want to be able to continue to access a local copy of a file even if

ones site becomes isolated from the rest of the network because of line failures. This could be accomplished by assigning a high enough weight to ones copy so that ones site itself would be enough to achieve a weighted majority. Many other policies can be implemented by a proper weight assignment to voting sites of a given site.

6.3.3 - File Policy Manager (FPM)

The File Policy Manager (FPM) is primarily responsible for implementing a distributed file system and also for implementing a secure information flow model. The system software above the FPM as well as all of the user software should be able to refer to files via virtual file names, which are netwide unique and location independent.

A virtual file or simply a file can be considered as a collection of virtual pages (the data pages) plus some mappings which define the file. One of these mappings specifies the set of virtual pages which compose a given file. This mapping is called the File to Virtual Page Mapping or F \rightarrow VP mapping. Before a file can be accessed at a given site, the F \rightarrow VP mapping for the file must be made available at the site. This operation is functionally equivalent to the operation of opening a file in centralized file systems. The MCM is used to lock the file in the desired mode. Once this is done, the FPM is able to request that the F \rightarrow

VP mapping be brought from a remote site. At this point the file is said to be known at the site. It may be the case that none of the pages of the file are present locally when the user starts using it.

For those pages which are locally present and only for them, the FPM keeps a mapping which indicates the physical disk location associated to the virtual page. This Mapping is called the Virtual Page to Physical Page mapping or VP -> PP mapping.

When a page which is not present locally is referenced by a user, the page must be brought over from a foreign site. This is accomplished via the Network Wide Page Faulting (NWPF) mechanism.

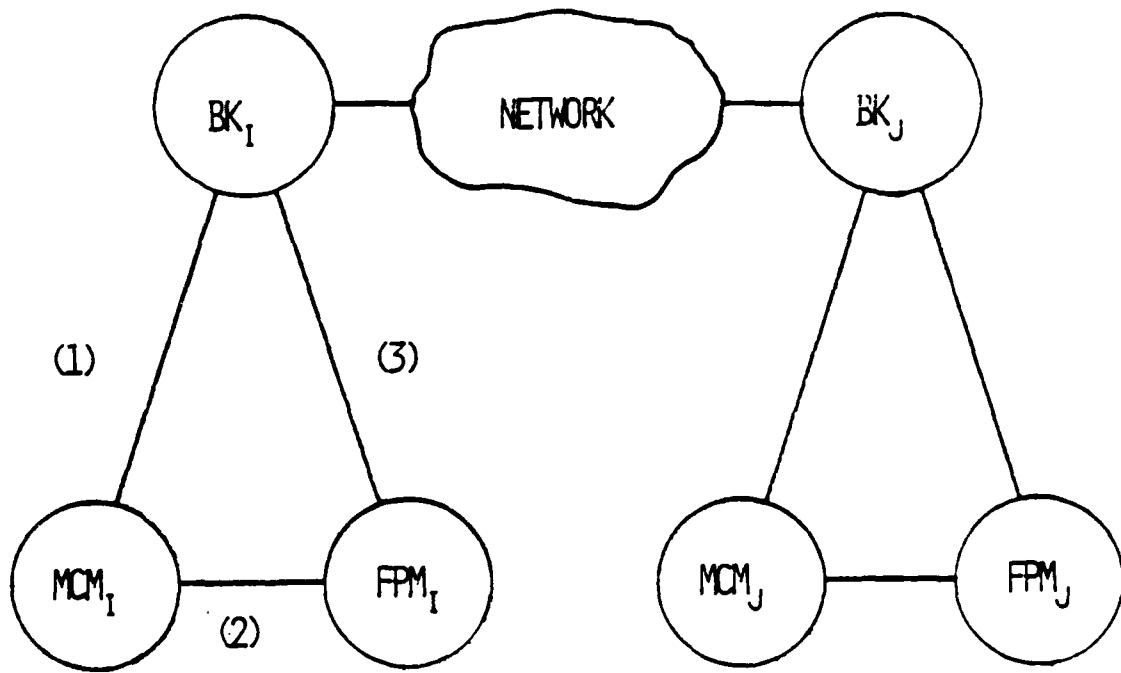
Network wide page faulting is implemented by the set of FPMs with the aid of the set of Base Kernels. When a user faults for a page which is not local, the FPM issues a SHIP IN call to the local Base Kernel requesting it to bring the desired VP from a site in the File to Site Mapping kept by the MCM. If the page cannot be found at the specified site, the FPM must choose another one and retry the operation until either the page is found or until the list of sites is exhausted. In the latter case, the page is declared to be lost and the file is declared to be in error. This error is reported to the Recovery Manager which will start a crash recovery operation.

Figure 6.3 shows the possible interactions between the FPM, MCM and BK. The picture also shows the operations associated with each such interactions.

In addition to the above described functions, the FPM must perform some operations which are required by the Recovery Manager. These functions include the sensing of interaction edges which involve files and the reporting of these edges to the Recovery Manager, as well as the taking of historical versions of objects and the restoration of objects to previously generated historical versions when demanded by the Recovery Manager. These issues will be covered in the following section when the internals of the Recovery Manager will be discussed.

6.3.4 - Recovery Manager

This system module is responsible for implementing the error recovery facilities needed to support reliable distributed computing. These facilities are based on the crash recovery model developed in chapter 4. Consequently, the core of the Recovery Manager is devoted to the maintenance of a local subgraph of the global condensed history graph G_c and in carrying out the crash set calculation algorithms developed in chapter 5. In order to perform the above functions, the Recovery Manager needs to interact with the FPM and with the BK, as well as with other RMs. These interac-



- (1) interaction with other MCMs via networkwide IPC facility.
- (2) locking of files.
- (3) issue of SHIP IN and SHIP OUT calls; interaction with other FPMs via networkwide IPC facility.

Figure 6.3 - Possible Interactions Between the FPM, MCM and the BK.

tions are in the form of messages, the description of which is given in what follows.

Whenever a recovery policy module - which is not part of the DSSB - dictates that an historical version of an object should be taken, the RM is charged with the task of initiating this action. In particular, if an historical version of a file must be generated, then the following message is sent from the RM to the FPM.

message name: take_file_hv

message arguments: (filename, timestamp)

message description: the Recovery Manager requests that the FPM take an historical version of the file which has virtual name filename. The historical version will be a file itself and the name of this file is filename.timestamp where timestamp is the clock. Additionally, the RM updates its local subgraph to reflect the generation of the new historical version in the way described in section 5.3.1 of chapter 5.

Two basic alternatives for taking historical versions of processes can be considered. The first one consists in having a distinguished process charged with the task of taking historical versions of all the processes. This process should be trusted otherwise security could be compromised. Alternatively, each process could take an historical

version of itself. This is the approach taken in the DSSB. The following message is sent from the RM to the process in question.

message name: take_proc_hv

message argument: (processname, timestamp)

message description: likewise the previous message, the RM requests that the process called processname generates an historical version of itself and stores it in a file called processname.timestamp where timestamp is as before. As in the previous message, the condensed history graph is updated to reflect the new historical version.

Interaction edges can be detected at two points in the DSSB: in the FPM (for file operations) and in the BK (for inter process communication). These interaction edges have to be reported to the RM for the purpose of updating the condensed graph. For efficiency reasons, these reports may contain a batch of interaction edges instead of a single one. For the case of inter process communication between remote processes, each of the BKs involved will sense it and report the interaction edge to their local Recovery Managers.

The kind of file operations that we are considering for the purposes of sensing and recording interaction edges are open and close operations. Let us examine how these in-

teraction edges are applied to the appropriate local subgraphs. Let P be a process requesting an operation on file F . As we mentioned in chapter 5, the subgraph $SG(P)$ associated with process P is located at the site where P is running. The file operation requested by P is sensed by the local FPM which passes it to the local RM. The Recovery Manager then adds the interaction edge to $SG(P)$ as described in section 5.3.2 of the previous chapter. We also require that the FPM local to the site where $SG(F)$ is currently located be involved in the operation. The most natural thing to do is to acquire the $F \rightarrow VP$ mapping from $location(F)$ - the site where $SG(F)$ is currently located. Now, by the assumption made in chapter 5, $SG(F)$ is located at the site where F is currently being accessed for update or where it was last used for update. Therefore, the $F \rightarrow VP$ mapping at $location(F)$ is the most current copy of it. The FPM at $location(F)$ senses the operation and passes the corresponding interaction edge to its local RM.

In any event, the RM will receive reports of interaction edges from its local BK and FPM. These reports come in the message described below.

message name: ie_report

message arguments: (report, message_seq_#)

message description: this message contains a batch of interaction edges. This batch, called report, has a message_seq_# assigned to it. This number is used to identify the acknowledgement message which must be sent by the RM to the sender of the ie_report.

The ie-report message must be acknowledged to its sender so that it can discard the report. This is done with the message:

message name: ie_report_ack

message argument: (message_seq_#)

message description: upon receipt of this message, the interaction edge report associated with message_seq_# is discarded.

Another operation on the graph is discarding of historical versions. As mentioned earlier, historical versions of files and processes are files like any other file as far as the FPM is concerned. When the RM decides that a given historical version must be discarded , it applies the appropriate collapsing operation in the graph (as described in section 5.3.3 of chapter 5) and requests the FPM to delete the file which contains the historical version. This is done via the message from the RM to the FPM.

message name: discard_hv

message argument: (hv_name)

message description: upon receipt of this message, the FPM will delete the file hv_name which contains an historical version of an object.

When a crash is detected the RM at the site where the error is detected will start the crash set calculation described in the previous chapter. The RMs at foreign sites will cooperate, when necessary by the protocol, in performing the crash set calculation.

Once the crash set calculation is over, the objects in the crash set must be restored to the historical versions indicated in the crash set. The restoration itself is done by the FPM upon request of the RM. The request has the form of the following message.

message name: restore_object

message arguments: (object_type, hv_name, message_seq_#)

message description: upon receipt of this message the FPM restores the object of the type specified by object_type (i.e. file or process) to its historical version in the file hv_name. The message_seq_# is used to associate this message with a reply to it containing the outcome of the restoration operation.

The Recovery Manager is notified of the result of the restoration operation via the following message.

message name: restoration_outcome

message argument: (outcome, message_seq_#)

message description: the outcome of the object restoration operation associated with message_seq_# is sent back to the RM. This outcome may either be: successful or unsuccessful in which case the specified historical version was not found.

Notice that in most of the operations just described, the RM interacts with its local FPM irrespective of the location of the desired file. This is possible because the FPMs implement a distributed file system which makes the location of a file transparent to software above the FPM.

6.4 - Conclusion

Large and important classes of applications of distributed computing require the existence of a secure and reliable base. Office automation systems and database management systems are examples of these classes of systems. The UCLA Distributed System Base (DSSB) described in this chapter provides the required base for such applications.

Six important architectural principles were singled out here, namely: network independent object names, the ex-

istence of multiple copies of objects, error confinement, minimization of the data critical for correct error recovery, separation of security from recovery relevant mappings and separation of mechanisms from policy issues.

The actual system architecture, which is composed of four major modules was described in this chapter. The architecture is layered in such a way that one level only needs the operations provided by the levels below for its correct operation. The set of Base Kernels implements a distributed process name space along with a network wide inter process communication mechanism. At the next level, we find the Multiple Copy Managers which implement the locking protocol necessary to coordinate access to multiple copies of files. At the next level, the File Policy Managers implement a network wide file name space and a distributed file system. The FPMs do not have to be concerned with the existence of multiple physical realizations of files since this issue is taken care by the MCMs. Finally, at the outermost level we find the Recovery Manager which is in charge of carrying out crash recovery operations.

CHAPTER 7
CONCLUSIONS AND
SUGGESTIONS FOR FUTURE RESEARCH

7.1 - Introduction

In this dissertation we made contributions in two areas, namely the areas of distributed systems and that of distributed databases. Besides summarizing these contributions, this chapter presents some suggestions for future research.

7.2 - Contributions to the Area of Distributed Systems

The design of distributed systems involves important issues such as:

- (a) interconnection structures
- (b) error recovery
- (c) network operating systems
- (d) security
- (e) program and data assignment

In this dissertation we made contributions to the problems of error recovery and of the design of network operating systems.

1.2.1 - Error Recovery

A formal model for crash recovery in computer systems was developed here. The model, which is in the form of a graph called the condensed history graph, considers the existence of a set of snapshots or historical versions for each recoverable object in the system as well as the records of flow of information between these objects.

This graph is defined in such a way that the following properties hold:

- a. any cutset of the graph containing branches associated with historical versions of distinct objects is a recovery set, i.e. a set of historical versions such that a global consistent state of the system is obtained if state restoration is done according to the historical versions indicated by the recovery set.
- b. the latest possible recovery set, called a crash set, was characterized in terms of a specially defined cutset of this graph. This cutset can be found using an efficient algorithm which is linear

in space and in time.

- c. all the historical versions associated with branches in a cycle of the condensed history graph are useless because they cannot participate of any crash set. Therefore they can be discarded.
- d. records of information flow between objects do not need to be stored anymore for crash recovery purposes once their effect has been taken into account in the graph. This fact is extremely important not only because it minimizes the storage requirements for the system but because the information flow pattern is crucial for correct error recovery. Therefore, if records of information flow needed to be stored, extremely reliable mechanisms would be needed for this purposes.

In a distributed system we cannot assume anymore the existence of a single complete version of the condensed history graph. Moreover, we cannot assume that the graph is being updated continuously. Instead, the graph is assumed to be partitioned into local subgraphs distributed among the several sites of the network. A protocol to update these local subgraphs as well as a protocol to do crash set calculation in a distributed manner was developed in this dissertation. The latter explores all the possible parallelism

which exists due to the fact that the graph is partitioned and that there are several sites which may be concurrently doing their part in the crash set calculation.

1.2.2 - User Interface

The user interface to a distributed system should be one in which the distributed nature of the system is not apparent. Examples of such interfaces are operating systems, database management systems and office automation systems. At any rate, the task of designing and building such interfaces is extremely simplified if they can be built on top of a distributed system base which makes the network transparent.

This dissertation contributed to the architecture of the UCLA Distributed Secure System Base (DSSB). The DSSB is a secure and reliable system base which utilizes the recovery protocols mentioned above. The DSSB is essentially composed of four modules, namely:

1. The BASE KERNEL which implements the security control mechanisms and also implements a network wide interprocess communication facility.
2. The MULTIPLE COPY MANAGER which is responsible for coordinating access to the several existing copies of the same file.

3. The FILE POLICY MANAGER which implements a distributed file system.
4. The RECOVERY MANAGER which implements the crash recovery facilities of the system.

The following architectural principles were identified in the design of the DSSB.

- a. Network Independence
- b. Multiple Copies
- c. Error Confinement
- d. Minimization of Recovery Relevant Data
- e. Separation of Security Relevant from Recovery Relevant Mappings
- f. Separation of Mechanisms from Policy

7.3 - Contributions in the Area of Distributed Databases

Major issues in the design of distributed databases are: data integrity, crash recovery, concurrency control and synchronization protocols. This dissertation presented a locking protocol to coordinate access to a distributed database and to maintain the database consistency. The protocol

is robust in the face of additional failures. Recovery is done in such a way that a maximum forward progress is achieved by the crash recovery procedures. The performance of the protocol does not degrade operations. In particular, the delay and communications cost incurred by an update request are not a function of the size of the network.

An undesirable side effect of locking is the problem of deadlocks. It was discussed in this dissertation that deadlock detection is superior than any other method for handling deadlocks in a distributed database environment. Deadlock detection requires the ability to detect cycle in a "state graph". In this dissertation we presented two algorithms to detect deadlocks in a distributed database - one is hierarchically organized and the other is distributed. Neither one of them requires that the whole graph be built at one site in order for deadlocks to be detected. These protocols are far superior than centralized deadlock detection approaches when the communications cost is a function of the distance.

7.4 - Suggestions for Future Research

The design of distributed systems is an area of research in its early stages. Some of the problems are understood, some are solved and some still have to be solved. In this dissertation we covered some of the problems and

developed formal models to aid in solving them. While we treated the problems of crash recovery using backward error recovery strategies, we did not address the problem of error detection. This problem is extremely important and no general solution to it has been found so far. An approach to error detection at the programming language level was presented in [ANDE 76]. This method suggests that more than one version of an algorithm be independently programmed. Each version is called an alternate block. The set of alternate blocks for a given algorithm is called a recovery block. Recovery blocks are nested. At the end of each alternate block there is an acceptance test. If the test succeeds then the execution of the associated alternate block is considered to be successful. If however the test fails an error is detected. In this case, the system is backed up to its state at the beginning of the alternate block in question. The next alternate block for the recovery block is attempted until either one of them succeeds or until all of them have been tried. The error detection problem is very important and deserves further research.

As for further research in the area of distributed databases we believe that suitable models of synchronization protocols need to be developed. These models should aid in the task of formally verifying the properties that one requires of these protocols. Formal verification clearly re-

quires a precise representation of a protocol where its properties can be formally stated. Graph models of parallel programs are a natural choice for protocol specification. Some of these models, like Petri Nets, limit themselves to modeling the flow of control of a computation and therefore, may strongly impair one's capability of using them as an aid in proving properties, like consistency, which deal with data semantics.

Other models, like Keller's model [KELL 76], extend Petri Nets by attaching to each transition an action on a set of variables which is executed if the transition would fire according to Petri Net rules and if a predicate attached to the transition is true. A modified version of this model has been used by Bochman [BOCH 77] in the specification and formal verification of communications protocols.

Ellis suggested in [ELLI 77] a different approach in which a production system, called L-system, is used to obtain a representation of a protocol. This representation includes flow of control as well as the semantic of a parallel computation in a uniform manner. The examination of the above mentioned methods to modelling of synchronization protocols in distributed databases is a subject worthwhile of further investigation.

Another area for further research is the area of performance analysis of synchronization protocols.

AD-A186 683

SECURE DISTRIBUTED PROCESSING SYSTEMS(U) CALIFORNIA
UNIV LOS ANGELES SCHOOL OF ENGINEERING AND APPLIED
SCIENCE G J POPEK DEC 78 UCLA-ENG-7955

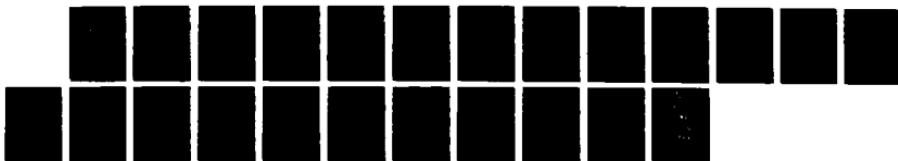
4/4

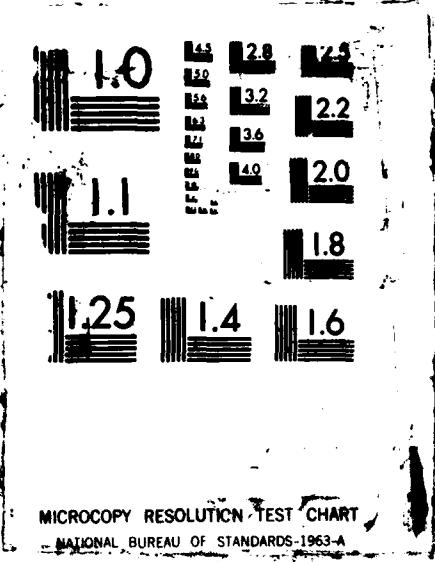
UNCLASSIFIED

MDA903-77-C-0211

F/G 12/7

NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

models of such protocols should take into account the transaction parameters such as interarrival time distributions at each node, distribution of the number of data items required at each of them, amount of processing time required and the like. With the use of queueing models and or simulation models one should be able to obtain results such as average response time, average communications cost as a function of the input parameters. Such an analysis would be an important tool for the database designer since it would help him understand the tradeoffs involved in the use of various synchronization protocols. While some work has been done in the area (see GARC 78 and GELE 78) much more needs to be done.

BIBLIOGRAPHY

- ALSB 76 Alsberg, P. A., G. Belford, J.D. Day, E. Grapa, "Multi-Copy Resiliency Techniques," Center for Advanced Computation, University of Illinois at Urbana-Champaign, CAC Document Number 202, May 31, 1976.
- ANDE 76 Anderson, T. and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 13-15, 1976, pp. 447-457.
- BADA 78 Badal, D.Z. and G.J. Popek, "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 273-285.
- BENJ 76 Benjamin, A. J., "Improving Information Storage Reliability Using a Data Network," Laboratory for Computer Science, MIT/LCS/TM-78, October 1976.
- BERN 77 Bernstein, P.A., N. Goodman, J.B. Rothnie, C.A. Papadimitriou, "Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)," Tech. Rep. # CCA-77-05, Computer Corporation of America, Cambridge, Massachusetts, June 15, 1977.
- BJOR 73 Bjork, L. A., "Recovery Scenario for a DB/DC System," Proceedings of the ACM National Conference, 1973.
- BOCH 77 Bochman, G. V. and J. Gecsei, "A Unified Method for the Specification and Verification of Protocols," Information Processing 77, North-Holland Publishing Company, 1977.
- CHAM 74 Chamberlin, D., R. F. Boyce and I.L. Traiger, "A Deadlock Free Scheme for Resource Locking in a Data-Base Environment," Proceedings of the IFIP

Congress, 1974.

- COFF 71 Coffman Jr., E.G., M.J. Elphick and A. Shoshani, "System Deadlocks," Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78.
- CROC 75 Crocker, S.D., "The National Software Works: A New Method for Providing Software Development Tools Using the Arpanet," presented at Consiglio Nazionale delle Richerche Instituto Di Elaborazione Della Informazione Meeting on 20 Years of Computer Science, Pisa, Italy, June 1975.
- DAVI 73 Davies, C. T., "Recovery Semantics for a DB/DC System," Proceedings of the ACM National Conference, 1973.
- DENN 76 Denning, D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, 1976.
- ELLI 77a Ellis, C. A., "A Robust Algorithm for Updating Duplicate Databases," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, California, May 25-27, 1977.
- ELLI 77b Ellis, C. A., "Consistency and Correctness of Duplicate Databases," Xerox Palo Alto Research Center Technical Report, 1977.
- ENSL 78 Enslow Jr., P.H., "What is "Distributed" Data Processing System?," Computer, January 1978, pp. 13-21.
- ESWA 76 Eswaran, K.P., J.N. Gray, R.A. Lorie and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," Communications of the ACM, November 1976.
- FARB 72b Farber, D.K. Larson, "The System Architecture of the Distributed Computer System - The Communications System," Proceedings of the Symposium on Computer Communication Networks and Teletraffic, New York, 1972.
- FARB 72c Farber, D. J., F. R. Heinrich, "The Structure of a Distributed Computer System," Proceedings of the International Conference on Computer

- Communications, Washington, 1972.
- FARB 72 Farber, D. J., K. C. Larson, "The Structure of a Distributed Computing System - Software," Proceedings of the Symposium on Computer Communication Networks and Teletraffic, New York, 1972.
- FRAN 71 Frank, H., I.T. Frisch, Communication, Transmission, and Transportation Networks, Addison-Wesley, Reading, Massachusetts, 1971.
- GARC 78 Garcia-Molina, H., "Performance Comparison of Two Update Algorithms for Distributed Data-bases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 108-119.
- GELE 78 Gelenbe, E. and K. Sevcik, "Analysis of Update Synchronization for Multiple Copy Data-Bases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 69-90.
- GRAY 76 Gray, J. N., R.A. Lorie, G.R. Putzolu and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," Modelling in Data Base Management Systems, G.M. Nijsen (ed.), North Holland Publishing Company, 1976.
- GRAY 78 Gray, J.N., "Notes on Data Base Operating Systems," Operating Systems: an Advanced Course, Springer-Verlag Berlin Heilderberg, 1978, pp. 394-481.
- HABE 69 Haberman, A.N., "Prevention of System Deadlocks," Communications of the ACM, Vol 12, No. 7, July 1969, pp. 373-377.
- HOPW 73 Hopwood, M.D., D. C. Loomis, L. A. Rowe, "The Design of a Distributed Computing System," University of California, Irvine, Technical Report #25, June 1973.
- KAMP 77 Kampe, M., C.S. Kline, G.J. Popek and E. Walton, "The UCLA Data Secure UNIX Operating System Prototype," Computer Science Department, School of Engineering and Applied Science, Technical Re-

port, July 1977.

- KELL 76 Keller, R.M., "Formal Verification of Parallel Programs," Communications of the ACM, Vol. 19, No. 7, July 1976.
- KEMM 79 Kemmerer, R., "Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation and Proof," Ph.D. Thesis in Computer Science, Computer Science Department, University of California, Los Angeles, January 1979.
- KING 73 King, P.F. and A.J. Collmeyer, "Database Sharing - Efficient Mechanism for Supporting Concurrent Processes," AFIPS Conference Proceedings, 1973 National Computer Conference.
- KLEI 70 Kleinrock, L. "Analytic and Simulation Methods in Computer Network Design," AFIPS Conference Proceedings, 1970 Spring Joint Computer Conference, 1970, Vol. 36, pp. 569-579.
- LAMP 76 Lampson, B. and H. Sturgis, "Crash Recovery in Distributed Data Storage Systems," Xerox Palo Alto Research Center Technical Report, 1976. (also to appear in the Communications of the ACM).
- LELA 78 LeLann, G., "Algorithms for Distributed Data-Sharing Systems Which Use Tickets," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978.
- MANN 74 Manning, E. and R. Peebles, "A Homogeneous Network for Data Sharing - Communications," CCNG report # E-12, University of Waterloo, March 1974.
- MANN 75 Manning, E. and R. Peebles, "Segment Transfer Protocol for a Homogeneous Computer Network," Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, Santa Monica, 1975.
- MENA 77 Menasce, D.A., G.J. Popek and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Systems," Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, UCLA-ENG-7808, SDPS-77-001 (DSS MDA 903-77-C-_211),

October 1977.

- MENA 78a Menasce, D.A., G.J. Popek and R. R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," Proceedings of the 1978 ACM/SIGMOD International Conference on the Management of Data, Austin, Texas, May 1978, pp. 1-14.
- MENA 78b Menasce, D.A. and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," Proceedings of the Third Berkely Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 215-232. (Also to appear in the IEEE Transactions on Software Engineering).
- MENA 78c Menasce, D.A., G.J. Rudisin, G.J. Popek and C.S. Kline, "A Proposed Architecture for the UCLA Distributed Secure System Base(DSSB)," UCLA Technical Report, November 1978.
- MENA 79 Menasce, D.A., R.R. Muntz and G.J. Popek, "A Formal Model of Crash Recovery in Computer Systems," Proceedings of the Twelfth Hawaii International Conference on System Science, Hawaii, January 4-5, 1979.
- METC 76 Metcalfe, R. and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Communications of the ACM, July 1976.
- PEEB 76 Peebles, R. and E. Manning, "A Computer Architecture for Larrge (Distributed) Data Bases," Proceedings of the Very Large Data Base Conference, 1976.
- PEEB 78 Peebles, R., "System Architecture for Distributed Data Management," Computer, January, 1978, pp. 40-47.
- POPE 74 Popek, G.J., and C.S. Kline, "Verifiable Secure Operating System Software," AFIPS Conference Proceedings, 1974 National Computer Conference, pp.145-151.
- POPE 78a Popek, G.J. and C.S. Kline, "Design Issues for Secure Computer Networks," Operating Systems: An Advanced Course, Springer-Verlag, Berlin Heil-

- derberg, 1978, pp. 517-546.
- POPE 78b Popek, G.J. and D.A. Farber, "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, September 1978, Vol. 21, No. 9, pp. 737-749.
- RAND 75 Randell, B., "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, June 1975, Vol. SE-1, No. 2, pp. 220-232.
- RAND 77 Randell, B., P.A. Lee and P.C. Treleaven, "Reliable Computing Systems," Computing Laboratory, University of Newcastle Upon Tyne, Technical Report Series, No. 102, May 1977.
- ROBE 70 Roberts, L.G. and B.D. Wessler, "Computer Networks to Achieve Resource Sharing," AFIPS Conference Proceedings, 1970 SJCC, Vol. 35, pp. 543-549.
- ROSE 77 Rosenkantz, D.J., R.E. Stearns and P.M. Lewis, "A System Level Concurrency Control for Distributed Database Systems," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, California, May 25-27, 1977.
- ROSE 78 Rosenkrantz, D. J., "Dynamic Database Dumping," Proceedings of the 1978 ACM/SIGMOD International Conference on the Management of Data, Austin, Texas, May 31- June 2, 1978, pp. 3-8.
- ROTH 77 Rothnie, J.B., N. Goodman, P.A. Bernstein, "The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case), Computer Corporation of America, Technical Report # CCA-77-02, Cambridge, Massachusetts, June 15, 1977.
- ROWE 73 Rowe, L. A., M.D. Hopwood, D. Farber, "Software Methods for Achieve Fail-Soft Behavior in the Distributed Computing System," Record 1973 Computer Software Reliability, IEEE, April 30 -

May 2, 1973.

- ROWE 75 Rowe,L.A., "The Distributed Computing Operating System," University of California, Irvine, Technical Report #66, June 1975.
- RUDI 79 Rudisin, G.J., "A Secure Reliable Distributed System Architecture," M.S. Thesis in Computer Science, Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, February, 1979.
- SHOS 70 Shoshani, A., and E.G. Coffman, "Sequencing Tasks in Multi-process, Multiple Resource Systems to Avoid Deadlocks," Proceedings of the Eleventh Annual Symposium on Switching and Automata Theory, October 1970, pp. 225-233.
- SHOS 78 Shoshani, A., Private Communication, Lawrence Berkeley Laboratory, July 1978.
- STEA 76 Stearns, R.E., P.M. Lewis II and D.J. Rosenkrantz, "Concurrency Control for Database Systems," Proceedings of the Symposium on Computer Science Foundations, 1976.
- STON 75 Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modifications," Electronics Research Laboratory, Memorandum No. ERL-M514, College of Engineering, University of California, Berkeley, March 17, 1975.
- STON 76 Stonebraker,M. and E. Neuhold, "A Distributed Data Version of Ingres," Electronics Research Laboratory, University of California,Berkeley,Memorandum # ERL - M612,September 1976.
- STON 78 Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, California, August 29-31, 1978, pp. 235-258.
- SWAN 77 Swan, R.J., S.H. Fuller and D.P. Siewiorek, "Cm* - a Modular, Multi-Microprocessor," AFIPS Conference Proceedings, 1977 National Computer

Conference, Vol. 46, pp. 637-xxx.

- TARJ 72 Tarjan, R.E., "Depth-First Search and Linear Graph Algorithms," SIAM Journal on Computing, June 1972, Vol. 1, No. 2, pp. 146-160.
- THOM 73 Thomas, R.H., "A Resource Sharing Executive for the Arpanet," AFIPS Conference Proceedings, 1973 SJCC, Vol. 42, pp. 155-163.
- THOM 76 Thomas, R. H., "A Solution to the Update Problem for Multiple Copy Data Bases which Uses Distributed Control," Bolt Beranek & Newman Technical Report No. 3340, July 1976.
- ZAFI 77a Zafiroplou, P., "Protocol Validation by Duologue-Matrix Analysis," IBM Zurich Research Laboratory, RZ 816(#27758), March 11, 1977.
- ZAFI 77b Zafiroplou, P., "A New Approach to Protocol Validation," IBM Zurich Research Laboratory, RZ 824(#28047), April 25, 1977.

APPENDIX A
PROOFS FOR ASSERTIONS IN CHAPTER 3

PROOF FOR ASSERTION 3.1: Let k be the number of relevant LLCs in the component and let $S = (X; x_1, x_2, \dots, x_k)$ be a global feasible state. The validity of this assertion can be readily verified by examination of Table 1. If a lock is in the LOCK table of the LC the state X is equal to $(1, 0, 0)$ if there are no pending lock release requests associated with the lock in question. But for $X = (1, 0, 0)$, the only possible LLC states are $(0, 1, 0)$ and $(1, 0, 0)$. Therefore the lock is present at every site and the assertion is proved.

PROOF FOR ASSERTION 3.2: Analogously to the proof of assertion 3.1, this assertion is easily proved by examination of Table 1. The only possible LLC states associated with the LC state $X = (0, 0, 0)$ are $(1, 0, 1)$ and $(0, 0, 0)$. Therefore the release request in question is present at every site and the assertion is proved.

PROOF FOR ASSERTION 3.3: We need to show that at the end of the notification phase, every site will end up with the same value for the trial list T and consequently the same value for NEWLC. Before doing so, we must prove the following two assertions.

Assertion A.1: Given two trial sequences, one of them must be a prefix of the other. In other words, let $X = x[1],$

$x[2], \dots, x[p]$ and $y = y[1], y[2], \dots, y[p], y[p+1], \dots, y[q]$ then $x[i] = y[i]$ for $i = 1, \dots, p$.

Proof: This assertion follows directly from the fact that the nomination is done following the nomination order and starting at the node that follows the crashed LC in this order.

Assertion A.2: The sequence, $T[1], T[2], \dots, T[m]$ of trial sequence values taken by the trial list T is such that if $T[i] = i[1], i[2], \dots, i[p]$, then $T[i+1] = i[1], i[2], \dots, i[p], i[p+1], \dots, i[p+q]$ for $i = 1, \dots, (m-1)$. In other words, $T[i]$ and $T[i+1]$ have the same prefix, namely $i[1], \dots, i[p]$.

Proof: By assertion A.1, either $T[i]$ is a prefix of $T[i+1]$ or vice versa. If $T[i]$ is empty this assertion is trivially true. So, from this point on assume that $T[i]$ is not empty. Assume now that this assertion is not valid. Therefore, it must be the case that if $T[i+1] = j[1], \dots, j[p]$ then $T[i] = j[1], \dots, j[p], j[p+1], \dots, j[p+q]$ for $q > 0$. But, as $T[i]$ is not empty, in order for T to be set to the value of $T[i+1]$, it must be the case that, during execution of the algorithm in section 3.2.3.1, $j[p+1]$ was not in $T[i]$. This contradicts our assumption and thus proves assertion A.2.

We can now prove assertion 3.3. Assume that this assertion is not true. So, there are at least two sites i and j for which its T values are different at the end of the notification phase. Let $T[i] = X, i[1], \dots, i[p]$ and $T[j] = Y$ where X is a common prefix. If $T[i]$ has the value X, $i[1], \dots, i[p]$ and the notification phase has already ended, then a NA message with trial sequence equal to $T[i]$ must have passed through node j. By that time, the algorithm in section 3.2.3.1 would have changed the value of $T[j]$ to that of $T[i]$. This contradicts our assumption and shows that all T and consequently NEWLC values will end up being the same at the end of the notification phase. That the value for NEWLC is the identification number of the latest LC to be nominated follows directly from assertion A.2.

PROOF FOR ASSERTION 3.4: In order to prove this assertion we must consider all the possible global states in which the set of LLCs can be left at when a crash occurs. Table 1 gives us the set of possible LLC states for each LC state. Therefore, when a crash occurs, the resulting global state is one of the possible combinations of LLC states associated with the state of the crashed LC just before the crash.

The list of all the possible combinations was derived from Table 1 and the eleven resulting cases are listed below. Each case is represented by a set S of LLC states where the set of global states associated with S is such

that there is at least one LLC state associated with each element of S.

| | |
|-------------------------------------|----------------------------|
| CASE 1: (0, 0, 0) | CASE 2: (1, 0, 0) |
| CASE 3 (0, 1, 0) | CASE 4: (1, 0, 1) |
| CASE 5: (0, 0, 0; 0, 1, 0) | CASE 6: (0, 1, 0; 1, 0, 0) |
| CASE 7: (0, 0, 0; 1, 0, 1) | CASE 8: (1, 0, 0; 1, 0, 1) |
| CASE 9: (0, 1, 0; 1, 0, 1) | |
| CASE 10:(0, 0, 0; 0, 1, 0; 1, 0, 1) | |
| CASE 11:(0, 1, 0; 1, 0, 0; 1, 0, 1) | |

The actions taken in each of the above cases by the algorithm that builds the sets L and R are summarized in

| CASES | ACTION |
|-----------|--|
| 1 & 2 | no action |
| 3,5 & 6 | add lock to the set L |
| 4,7 & 8 | add lock to the set R |
| 9,10 & 11 | If seq#(0,1,0) > seq#(1,0,1) Then add lock to the set L Else add lock to the set R |

Table A.1 - Actions taken by the algorithm that builds the sets L and R.

Table A.1.

Assertion 3.4 states that a lock p is either in all the the LOCK tables of S(p) or it is in none of them. Let us prove this statement as a lemma.

Lemma: At the end of the LCR mechanism the following is true. If the lock p is in one LOCK table of a site in S(p) then it must be in the LOCK table of all of them.

Proof: Assume that it is not in at least one LOCK table but that it is in some of them. Then, this lock was not included in the set L otherwise it would have been added to all the LOCK tables. In a similar vein, it was not included in the set R otherwise it would have been deleted from all the LOCK tables. Therefore, the lock was in no L-list nor R-list when the LCR mechanism started. This eliminates cases 3 through 11. We are then left with cases 1 and 2. Case 1 is also eliminated because by assumption the lock is in at least one LOCK table. Case 2 is also eliminated because by assumption the lock is not in at least one LOCK table. All the cases have been eliminated. This is a contradiction and proves the lemma.

The complete statement of assertion 3.4 now follows directly from the algorithm which builds the sets L and R. As indicated in Table A.1, the actions taken in each case are such that maximum forward progress is achieved in each case. In other words, locks that would have been granted are added to all the LOCK tables and locks that would have been released are deleted from all the LOCK tables. Thus, assertion 3.4 is proved.

APPENDIX B

ALGORITHM FOR DISTRIBUTED CRASH SET CALCULATION

An Algol-like description of the algorithm to do crash set calculation in a distributed system is given in this appendix.

Procedure CRASHSET is executed at the site where the error is detected. In order to make explicit the parallel computation which exists in this distributed algorithm we introduced some constructs to the language.

One of them is the COBEGIN statement. It is used to indicate that a message is sent to several sites causing a certain algorithm to be executed at those sites. The syntax of the COBEGIN statement is given below.

```
COBEGIN FOR j := a STEP b UNTIL c DO  
    <boolean expression> : A(p1, p2, ..., pn) ;  
COEND;
```

Its semantics is as follows. The variable *j* (called a site variable) is incremented according to the FOR clause. For each value of *j* the boolean expression is evaluated and if it is true a message with parameters *p₁, p₂, ..., p_n* is sent to site *j* causing algorithm *A* to be executed at that site.

Another construct is the REPLY declaration which may appear in the procedure heading and it is used to simulate the fact that some messages require a reply to the sender site. In particular, assume that site Si sends a message to site Sj which causes algorithm A to be executed at Sj. If as a result of this message and of A's execution a message containing certain parameters must be sent back to Si, these parameters must be declared as REPLY in the procedure body of A. Also, a reply variable must be assigned a value only once during the execution of algorithm A. When algorithm A returns, the reply message is considered to be sent to the calling site.

The procedure FCRASHSET is the one executed to continue the crash set calculation at a foreign site.

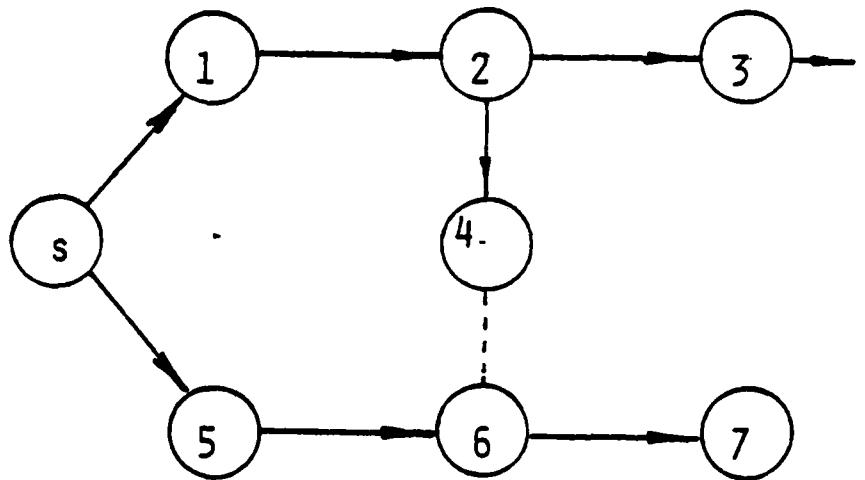
The DFS procedure differs from the one given in chapter 4 in two aspects.

- a. When examining the adjacency list of the node currently being visited the DFS procedure marks as output connections the non local nodes.
- b. As soon as we visit a node such that its label contains an object x then we know right away that the object x must be backed up although we may still not know which historical version should be used to restore it. We must at this point prevent

object x from interacting with other objects since otherwise the graph could grow in such a way that some objects which should be in the crash set end up not being in it. This operation is called "freezing" the object. Figure B.1 helps to illustrate the point. If in the graph of figure B.1.a the interaction edge between nodes 4 and 6 occurred after node 4 was visited in calculating $G^*(2)$ then we would obtain as a result the graph in figure B.1.b while the correct result for $G^*(2)$ is the graph in figure B.1.c .

Once the graph $G^*(X)$ has been found each of the sites which contributed to find $G^*(X)$ are requested to find the subset of the crash set local to them. This is indicated by sending a message to each of these sites and having them execute the FCALC procedure. This procedure basically finds the set of branches incident into nodes of the portion of $G^*(X)$ local to these sites and which are not in $G^*(X)$.

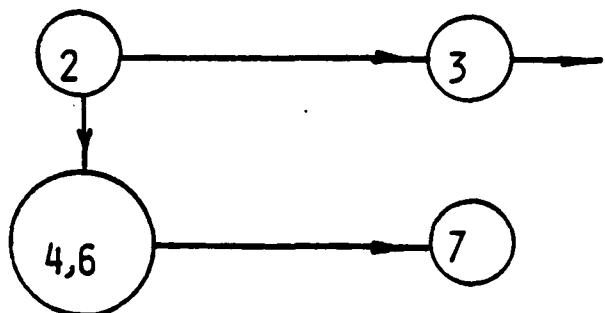
Finally, `crash_set(X)` is simply the union of all the partial crash sets obtained at each participating site.



(B.1.a)



(B.1.b)



(B.1.c)

Figure B.1 - Example to Illustrate the Need of the "Freeze" Operation.

```

PROCEDURE CRASHSET(X) ;
BEGIN
    COMMENT nodeset[j] = true if node j is in G*(X)
    nsets[k,*] is the buffer for sending and
                receiving nodeset to site k.
    last[j] = true if node j is lastnode for
                any object.
    processed[j] = true if node j is numbered &
                    last[j] = true.
    fsite[k] = true if there is an output connec-
                    tion to site k.
    site[j] = 0 if node j is local &
                = k if node j is local to site k.
    outconnection[j] = true if node j may be used
                    as a root in applying DFS at a
                    foreign site.

INTEGER i,j ;
INTEGER ARRAY site[1:n] ;
BOOLEAN ARRAY nodeset[1:n], last[1:n], processed[1:n],
            fsite[1:m], outconnection[1:n],
            nsets[1:m, 1:n] ;

PROCEDURE DFS(v,u) ;
BEGIN
    number[v] := i := i + 1;
    nodeset[v] := true ;

    COMMENT every object found to be in the crash
          set should be freezed . ;
    FOR EACH object z G label(v) DO freeze z ;

    FOR EACH y -> G into(v)
        DO S(y,X) := S(y,X) U {y -> v};
    FOR w in the adjacency list of v DO
        IF site[w] = 0 COMMENT w is local;
        THEN IF w is not numbered
            THEN DFS(w,v) ;
            ELSE ;
        ELSE IF nodeset[w] = false
            THEN BEGIN
                COMMENT w is an output
                      connection;
                outconnection[w] := true ;
                fsite[site[w]] := true ;
            END;
    END dfs;

```

```

COMMENT initialization phase;
i := 0;
processed := nodeset := outconnection := false;
fsite := false ;
crash_set(X) := Ø ;
FOR EACH node y in G1 do S(y,X) := Ø;

COMMENT apply DFS to local graph for each
object x ∈ X. ;
FOR EACH object x ∈ X DO
IF processed[lastnode(x)] = false
THEN DFS(lastnode(x),0);

COMMENT continue to apply depth first search at
foreign sites which have output connections. ;
FOR j := 1 STEP 1 UNTIL m DO
nsets[j,*] := nodeset;
COBEGIN FOR j := 1 STEP 1 UNTIL m DO
fsite[j] = true : FCRASHSET(outconnection,
nsets[j,*],j) ;
COEND ;

COMMENT nodeset is now obtained by "oring" all the
nodesets obtained at different sites. ;
FOR j := 1 STEP 1 UNTIL m DO
nodeset := nodeset V nsets[j,*] ;

COMMENT start final computation of crash set(X) locally
by taking the union of all S(y,X) for nodes
y ∈ G*(X). ;
FOR j := 1 STEP 1 UNTIL n DO
IF nodeset[j] = false
THEN crash_set(X) := crash_set(X) U S(j,X) ;
ELSE fsite[site[j]] := true ;

COMMENT continue final computation of crash set at
foreign sites
FOR j := 1 STEP 1 UNTIL m DO
crash_set(j,X) := crash_set(X) ;
COBEGIN FOR j := 1 STEP 1 UNTIL m DO
fsite[j] = true : FCALC(crash_set(j,X), nodeset) ;
COEND;

COMMENT complete crash set calculation by taking
the union of all the partial computations;
FOR j := 1 STEP 1 UNTIL m DO
IF fsite[j] = true
THEN crash_set(X) := crash_set(X) U crash_set(j,X);
END crashset;

```

```

PROCEDURE FCRASHSET(roots,nset,isite) ;
  VALUE roots, isite; REPLY nset;
  BOOLEAN ARRAY roots ;
  INTEGER isite ;
  BEGIN
    COMMENT nodeset[j] = true if node j is in
           the local subgraph.
    nsets[k,*] is the buffer for sending and recei-
           ving nodeset to site k.
    fsite[k] = true if there is an output connec-
           tion to site k.
    site[j] = 0 if node j is local &
           = k if node j is local to site k.
    outconnection[j] = true if node j may be used
           as a root in applying DFS at a
           foreign site.

    INTEGER i, j;
    INTEGER ARRAY site[1:n] ;
    BOOLEAN ARRAY nodeset[1:n], fsite[1:m],
           outconnection[1:n], nsets[1:m,1:n] ;
    COMMENT initialization phase
    i := 0 ; nodeset := nodeset V nset ;
    outconnection := false; fsite := false;
    FOR EACH node y in G1 DO S(y,X) := Ø ;

    COMMENT apply DFS starting at each not yet visited
          root ;
    FOR j := 1 STEP 1 UNTIL n DO
      IF roots[j] = true & site[j] = isite &
        nodeset[j] = false
      THEN DFS(j,0) ;

    COMMENT continue to apply DFS at foreign sites.
    FOR j:= 1 STEP 1 UNTIL m DO
      nsets[j,*] := nodeset;
      COBEGIN FOR j := 1 STEP 1 UNTIL m DO
        fsite[j] = true : FCRASHSET(outconnection,
                                      nsets(j,*), j) ;
      COEND ;

    COMMENT nodeset is now obtained by "oring" the nsets
          obtained at different sites;
    FOR j := 1 STEP 1 UNTIL m DO
      nodeset := nodeset V nsets[j,*] ;

    COMMENT return value of nodeset to calling site;
    nset := nodeset;
  END fcrashset ;

```

```
PROCEDURE FCALC(cset,nodeset) ;
  VALUE nodeset ; REPLY cset ;
  BOOLEAN ARRAY nodeset ;
  BEGIN
    INTEGER j ;
    crash_set(X) := cset ;
    FOR j:= 1 STEP 1 UNTIL n DO
      IF nodeset[j] = false
      THEN crash_set(X) := crash_set(X) U S(j,X) ;
    COMMENT return value of crash set to calling site;
    cset := crash_set(X) ;
  END fcalc ;
```

END
DATE
FILMED
JAN
1988